Ausgabe 03/2023

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 03

Java aktuell

iJUG Verbund
www.ijug.eu

Newcomer-Beiträge
AWS Serverless Lambda-Funktionen, Projektkommunikation, Mob Programming, Einstieg in die Softwareentwicklung

Spring Native
Java und Spring Boot für die Cloud, Testen von Kubernetes-Controllern und -Operatoren, Strings

Next Generation

Inhalt

Möglichkeiten zum Testen von AWS Serverless Lambda-Funktionen

Erfolgreiche Kommunikation im Projekt

Spring Native:
Mit Java und Spring Boot in die Cloud

Testen von in Java implementierten Kubernetes-Controllern und -Operatoren

Lokales Testen von AWS Serverless Lambda-Funktionen

Jens Knipper, OpenValue Düsseldorf

Das Testen von Microservices wird mittlerweile hinreichend praktiziert. Bei Serverless Functions wie AWS-Lambdas wird meist nur manuell nach einem Deployment getestet, speziell wenn es um die Logik geht, die durch AWS (Amazon Web Services) bereitgestellt wird. Dabei ist es auch möglich, die Anwendung automatisiert lokal hochzufahren und zu testen. Aber wie geht das in einem (relativ) geschlossenen System wie der AWS-Cloud? Durch den geschickten Einsatz von LocalStack, Testcontainers und dem AWS-SDK ist es unter anderem möglich, Component-Tests automatisiert auszuführen und etwaige Fehler in der Benutzung der Lambda-Funktion aufzudecken.

Als Dienstleister komme ich durch den Einsatz in unterschiedlichen Projekten auch zwangsläufig mit vielen verschiedenen Technologien in Berührung. Dabei lässt es sich nicht verhindern, dass auch neue Technologien darunter sind. Bei meinem aktuellen Projekt bin ich das erste Mal mit AWS Serverless Lambda-Funktionen in Kontakt gekommen und habe mich gefragt: Wie testet man das?

Da ich zuvor schon an Microservices gearbeitet und eine gewisse Ähnlichkeit zu Lambdas gesehen habe, habe ich versucht, eine Test-Strategie für Microservices anzuwenden. Das funktioniert in unserem Projekt so gut, dass wir unsere Dependency-Updates mittlerweile automatisieren und uns darauf verlassen können, dass wir beim Update von AWS-Dependencies mitbekommen, wenn etwas nicht mehr funktionieren sollte. Bevor wir tiefer in das Ganze einsteigen, schauen wir uns an, wie man Microservices testet.

Testen von Microservices

Zum Testen von Microservices gibt es bereits sehr gute Literatur *[1]*. Wir unterteilen die Tests in Unit-, Integration- und Component-Tests. Unit-Tests testen einzelne Komponenten wie die Domäne und die Service-Schicht. Wir nutzen Integration-Tests, um das Zusammenspiel mit externen Komponenten zu testen. In Component-Tests fahren wir unseren Microservice hoch und feuern Anfragen auf diesen, um die erhaltenen Ergebnisse zu verifizieren.

Ein konkretes Beispiel für einen Microservice und die Test-Struktur ist in *Abbildung 1* zu finden. Dieser Microservice ist nach dem Ports-and-Adapter-Pattern *[2]* aufgebaut. Alle externen Dienste sind durch einen Adapter mit dem Service-Layer verbunden.

Der REST-Adapter ist die äußere Schnittstelle der Anwendung. Sie gibt an, wie der Service aufgerufen werden kann und welche Werte zu erwarten sind. Der HTTP-Adapter stellt die Grenze, etwa zu einem anderen Service, dar. Der Service wird aufgerufen, um Werte zu erhalten, die wir gegebenenfalls für eine Berechnung benötigen. Der Datenbank-Adapter dient dazu, Werte aus der Datenbank zu speichern und zu lesen.

Die Kästen in *Abbildung 1* geben an, wo die Grenzen der einzelnen Tests liegen. Die kleinste Art von Test ist der Unit-Test. Mit ihm können wir die Module testen, die unabhängig von externen Anwendungen sind. Das sind die Domäne, die Service-Schicht und, je nach Framework, der REST-Adapter.

Integration-Tests testen die Integration zu externen Komponenten. In unserem Beispiel sind es ein anderer Service und eine Datenbank. Die Integration zur Datenbank können wir mit einem Tool wie Testcontainers oder einer In-Memory-Datenbank testen.

Testcontainers ist ein Framework, das in Tests genutzt werden kann, um automatisiert externe Komponenten in einem Docker-Container hochzufahren. Der komplette Lifecycle des Containers wird dabei von dem Framework verwaltet.

Um die Integration zu einem externen Service zu testen, können wir einen Mock-HTTP-Server wie WireMock nutzen. Wir nehmen unseren Client und ändern die Basis-Adresse, um Anfragen an den Mock-Server zu senden. Dieser gibt dann vordefinierte Werte zurück.

Component-Tests testen unseren Microservice als Ganzes, wobei externe Abhängigkeiten ersetzt werden. Wir ersetzen sie einfach wie in den Integration-Tests durch Mocks oder Ähnliches. Anschließend fahren wir in dem Test unsere Anwendung hoch und schicken Anfragen auf diese. Die zurückgegebenen Werte oder die in die Datenbank geschriebenen Werte vergleichen wir anschließend mit unseren Erwartungen.

Am Ende haben wir alle Elemente unseres Microservice getestet. Wenn

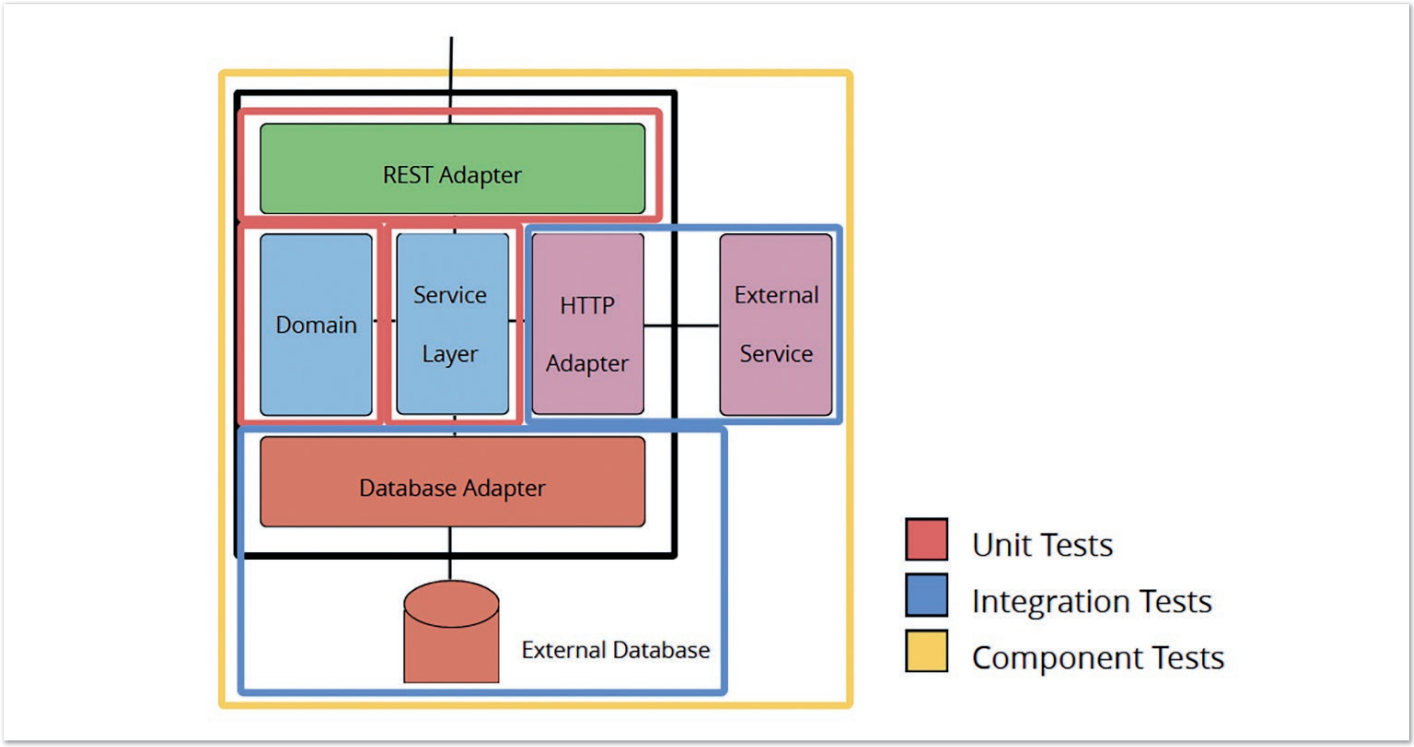

Abbildung 1: Aufbau eines Microservice mit Ports-and-Adapter-Pattern. Die farbigen Kästen geben an, wie die einzelnen Komponenten getestet werden können. (© Jens Knipper)

möglich, wurde versucht, die einzelnen Module isoliert zu testen. Die Integration mit externen Abhängigkeiten wurde getestet und dass der Service als Ganzes, inklusive des Zusammenspiels der einzelnen Komponenten, funktioniert, wurde mithilfe eines Component-Tests geprüft.

Als kleine Randnotiz: Würden wir jetzt alle Arten von Tests entsprechend ihrer Menge gruppieren, würden wir die Testpyramide erhalten.

Einführung in Lambda-Funktionen

AWS Serverless Lambda-Funktionen sind kleine, simple Funktionen, die ausgeführt werden können. Getriggert werden diese durch Events, die frei definiert werden können. Alternativ kann auch auf AWS-spezifische Events, wie ein Datei-Upload zu S3 (Simple Storage Service), zurückgegriffen werden.

Der Name Funktion ist dabei treffend. Es wird kein Webserver oder Ähnliches zum Ausführen benötigt. Es muss lediglich ein Interface implementiert werden, das eine Methode zum Behandeln des Events enthält. Das Ganze kann dann als zip, in Java als jar, zu AWS hochgeladen werden. Zudem gibt es noch weitere Möglichkeiten, ein Deployment durchzuführen, die eine bessere Automatisierung ermöglichen. Diese sollen hier aber nur Rande erwähnt werden.

Nach dem Upload kümmert sich AWS komplett um den Betrieb des Lambdas. Um Dinge wie Skalierung muss man sich keine Gedanken machen. Die Funktion wird bei Last automatisiert hochskaliert und bei weniger Events entsprechend herunterskaliert. Am Ende muss man nur die Laufzeit bezahlen. Bei wenig Last wird es so entsprechend günstiger.

In meinem aktuellen Projekt setzen wir Lambdas für asynchrone Tasks im Stile von „fire and forget" ein. Ein anderer Anwendungsfall ist das Rausziehen einzelner Funktionen aus Microservices, um eine bessere Skalierung zu ermöglichen.

Testen von Lambdas

Die Frage ist nun: Warum eignet sich der Test-Ansatz von Microservices auch für Lambdas? Mit Lambdas lassen sich tatsächlich auch Microservices bauen! Auch die klassische Architektur, das Ports-and-Adapter-Pattern, ist problemlos anwendbar. Im Gegensatz zu Microservices sind Lambdas allerdings in ihrer Größe beschränkt. Durch das von AWS bereitgestellte Framework sind die Beschränkungen offensichtlicher, beispielsweise kann ein Lambda immer nur ein bestimmtes Event verarbeiten. Das Verarbeiten unterschiedlicher Events ist nur über Umwege möglich.

Eine Teststrategie, wie bei Microservices, scheint also naheliegend. Das bedeutet auch, dass wir unsere bekannten Werkzeuge weiterverwenden können. Dazu gehören Dependency Injection, Testcontainers und WireMock. Da wir uns in einer neuen Umgebung befinden, kommen wir um neue Tools nicht herum. Im AWS-Umfeld ist LocalStack *[3]* eine besonders große Hilfe. LocalStack ist ein komplett lokal funktionierender Cloud-Stack, mit dem man Services und Lambdas von AWS lokal laufen lassen kann.

Beispiel-Use-Case

Die Umsetzbarkeit des Verfahrens lässt sich am besten anhand eines Beispiels erklären. Dafür stellen wir uns die folgende Situation vor: Wir möchten ein Thumbnail von einem Bild generieren, weshalb wir das Bild auf den AWS Filestorage S3 hochladen. Anschließend sprechen wir das Lambda an und befehlen ihm, ein Thumbnail daraus zu erstellen. Das Lambda lädt also das Bild von S3 herunter, skaliert es und schiebt das verkleinerte Bild anschließend wieder in den Filestorage. Als Rückmeldung bekommen wir vom Lambda die Information, wo wir das Bild herunterladen können.

Das ist eine recht simple Anwendung, also machen wir uns daran, diese umzusetzen. Wie bereits erwähnt, müssen wir ein Interface als Einstiegspunkt für das Lambda implementieren, um Events zu verarbeiten *(siehe Listing 1)*. Wir erstellen also das Interface, definieren Input und Output und implementieren die geforderte Methode. Generell versuche ich, diese Klasse so klein wie möglich zu halten, um einzelne Komponenten unabhängig vom Framework testen zu können. Die einzige Ausnahme ist der Konstruktor der Klasse, in dem die Services erstellt werden. Durch Nutzung von manueller Dependency Injection ist es am Ende möglich, die Komponenten isoliert voneinander zu testen. Bei komplexeren Lambdas kann der Konstruktor schon mal ein bisschen größer werden und eine baumartige Struktur annehmen, wobei der *EventHandlingService* die Wurzel bildet, die alle anderen Komponenten vereinigt.

Wir greifen nicht auf vorgefertigte Events zurück und definieren den Input und Output des Lambdas selbst *(siehe Listing 2)*. Als Input benötigen wir die Information, wo das Bild zu finden ist. Dafür reichen der Name des S3 Bucket und der Dateiname. Bei der Klasse handelt es sich um ein simples POJO mit Getter- und Setter-Funktionen, die der Übersicht halber hier ausgelassen werden. Records können zurzeit leider nicht genutzt werden, da maximal Java in Version 11 von AWS unterstützt wird. Analog dazu ist der Output. Auch hier finden wir den Namen des Bucket und den Dateinamen. Diese referenzieren allerdings das generierte Thumbnail.

Der *EventHandlingService* wird ausgeführt, nachdem das Lambda aufgerufen wurde. Hier ist die eigentliche Logik zu finden *(siehe Listing 3)*. Das zu verkleinernde Bild wird mit dem *S3Service* heruntergeladen und durch den *ImageService* verkleinert. Anschließend werden das Thumbnail hochgeladen und die Details, wo es zu finden ist, in Form des Outputs zurückgegeben. Auch hier ist der Code wieder ein bisschen vereinfacht. Zwischen den Schritten geschieht die Umwandlung zwischen *InputStream*, Bild und *OutputStream*. Dies ist für das Verständnis allerdings nicht weiter relevant.

Der *S3Service* ist recht simpel. Er nutzt das AWS-SKD, um Dateien hoch- und herunterzuladen. Ebenso der *ImageService*, der AWT-Funktionen benutzt, um das Bild auf eine festgelegte maximale Größe zu skalieren. Aber ehrlich gesagt kümmern uns die Interna der Services gar nicht – wir wollen sie nur testen! Dazu genügt es, die Inputs und zu erwartenden Outputs zu kennen.

Testen des Use-Case

Machen wir uns als Erstes an den Test für den *ImageService* *(siehe Listing 4)*. Der Service hat keine externen Abhängigkeiten und kann deshalb einfach per Unit-Test getestet werden. Wir initialisieren den Service einfach mit einer vorgegebenen maximalen Größe für die Thumbnails. Anschließend laden wir ein Bild aus dem Dateisystem, geben es dem Service und erwarten, dass das Bild auf die entsprechende Größe skaliert wurde. Wie der Service intern arbeitet, ist uns dabei egal, solange er die Bilder auf unsere vorgegebene Größe skaliert.

```
public class EventHandler implements RequestHandler<Input, Output> {
  private final EventHandlingService eventHandlingService;

  public EventHandler() {
    final S3Service s3Service = new S3Service(...);
    final ImageService imageService = new ImageService(...);

    this.eventHandlingService = new EventHandlingService(s3Service, imageService);
  }

  @Override
  public Output handleRequest(final Input input, final Context context) {
    return eventHandlingService.handleEvent(...);
  }
```

Listing 1: Aufbau des EventHandler zum Erstellen von Thumbnails

```
public class Input {
  private String bucket;
  private String key;

  ...
 }
```

Listing 2: Inhalt des eingehenden Events des Lambdas

```
public class EventHandlingService {
  private final S3Service s3Service;
  private final ImageService imageService;

  public EventHandlingService(final S3Service s3Service, final ImageService imageService) {
    this.s3Service = s3Service;
    this.imageService = imageService;
  }

  public Output handleEvent(final String bucket, final String fileKey) {
    final InputStream inputStream = s3Service.getObject(...);
    ...
    final BufferedImage newImage = imageService.resize(...);
    ...
    final String link = s3Service.uploadFile(...);
    return new Output(...);
  }
}
```

Listing 3: Aufbau des EventHandlingService. Hier ist die gesamte Logik der Anwendung zu finden

```
class ImageServiceTest {
  private final int maxDimension = 300;
  private final ImageService imageService = new ImageService(maxDimension);

  @Test
  void shouldResizeImage() throws IOException {
    // given
    final File testImage = new File("src/test/resources/image.png");
    final BufferedImage image = ImageIO.read(testImage);

    // when
    final BufferedImage resizedImage = imageService.resize(image);

    // then
    assertThat(resizedImage.getHeight()).isLessThanOrEqualTo(maxDimension);
    assertThat(resizedImage.getWidth()).isLessThanOrEqualTo(maxDimension);
  }
}
```

Listing 4: Unit-Test für den ImageService

```
@Testcontainers
class S3ServiceTest {
  @Container
  private static final LocalStackContainer localStack =
      new LocalStackContainer(DockerImageName.parse("localstack/localstack"))
          .withServices(LocalStackContainer.Service.S3)
          .withEnv("DEFAULT_REGION", Region.EU_CENTRAL_1.toString());

  private static final AwsCredentialsProvider credentialsProvider =
      StaticCredentialsProvider.create(
          AwsBasicCredentials.create(localStack.getAccessKey(), localStack.getSecretKey()));
  ...
  private final S3Service s3Service =
      new S3Service(
          localStack.getEndpointOverride(LocalStackContainer.Service.S3), credentialsProvider);

  @Test
  void getObjectShouldReturnInputStream() throws IOException {
    // given
    createBucketAndFile("bucket", "file", "content");

    // when
    final InputStream result = s3Service.getObject("second-bucket", "file");

    // then
    final String resultAsString = new String(result.readAllBytes());
    assertThat(resultAsString).isEqualTo("content");
  }
  ...
}
```

Listing 5: Integration-Test für den S3Service. AWS-spezifische Komponenten werden hier lokal durch LocalStack ausgeführt

```
@Testcontainers
public class ComponentIT {
  ...

  @Container
  private static final LocalStackContainer localStack =
      new LocalStackContainer(DockerImageName.parse("localstack/localstack"))
          .withServices(Service.LAMBDA, Service.S3)
          .withNetwork(Network.SHARED)
          .withNetworkAliases(localstackNetworkAlias)
          .withEnv("LAMBDA_DOCKER_NETWORK", ((NetworkImpl) Network.SHARED).getName())
          .withFileSystemBind(
              new File("target/").getPath(), "/opt/code/localstack/target/", BindMode.READ_ONLY);
  ...

  @BeforeAll
  public static void beforeAll() throws IOException {
    createLambdaFunction();
  }

  @Test
  void testHappyPath() throws IOException {
    // given
    createBucket("bucket");
    uploadFile("bucket", "image.png", "src/test/resources/image.png");

    // when
    final InvokeResponse response = invokeLambda("{\"bucket\": \"bucket\", \"key\": \"image.png\"}");

    // then
    assertThat(response.statusCode()).isEqualTo(200);

    final String resultAsJson = new String(response.payload().asByteArray());
    final BufferedImage resizedImage = getImage(resultAsJson);
    assertThat(resizedImage.getHeight()).isLessThanOrEqualTo(MAX_DIMENSION);
    assertThat(resizedImage.getWidth()).isLessThanOrEqualTo(MAX_DIMENSION);
  }
  ...
}
```

Listing 6: Component-Test des eigenen Lambdas, durch Nutzung von LocalStack

```
private static void createLambdaFunction() throws FileNotFoundException {
  final Map<String, String> variables =
      Map.of(
          ...
          "S3_ENDPOINT_OVERRIDE",
          "http://" + localstackNetworkAlias + ":" + 4566;

  final CreateFunctionRequest request =
      CreateFunctionRequest.builder()
          ...
          .handler("de.jensknipper.lambdatesting.EventHandler")
          .packageType(PackageType.ZIP)
          .code(
              FunctionCode.builder()
                  .zipFile(
                      SdkBytes.fromInputStream(
                          new FileInputStream("target/testing-aws-lambdas-1.0.jar")))
                  .build())
          .environment(Environment.builder().variables(variables).build())
          .build();

  final CreateFunctionResponse response = getLambdaClient().createFunction(request);
}
```

Listing 7: Deployment des Lambdas in LocalStack

Der *S3Service* ist ein wenig komplizierter zu testen. In *Listing 5* ist ein beispielhafter Test dargestellt. Hier haben wir eine Abhängigkeit zum Filestorage S3 und müssen somit auf einen Integration-Test zurückgreifen. Deshalb kommt eines der neuen Tools zum Einsatz: LocalStack. Zusammen mit Testcontainers fahren wir einen zu S3 kompatiblen Filestorage lokal hoch. Dem *S3Service* müssen wir nur die lokalen Daten mitteilen, damit er sich dagegen und nicht gegen die AWS-Cloud verbindet. Dazu dient der sogenannte *EndointOverride*. Für den Test erstellen wir uns noch einen eigenen S3Client, um Dinge wie Setup und Verifikation unabhängig von dem Service, den wir testen wollen, durchführen zu können. Mit dem Client erstellen wir dann einen Bucket und eine simple Datei mit Text als Inhalt. Diese versuchen wir anschließend mit dem zu testenden S3Service herunterzuladen. Am Ende verifizieren wir, ob die Datei erfolgreich heruntergeladen werden konnte, indem wir die Inhalte vergleichen.

Zuletzt widmen wir uns dem Component-Test. Unsere Intention ist es hier, das Lambda einmal zu erstellen, hochzufahren, eine Anfrage an dieses zu schicken und den Rückgabewert zu verifizieren. Der Code ist vereinfacht in *Listing 6* dargestellt. Da wir auch hier von AWS abhängig sind, ist es nötig, LocalStack in Kombination mit Testcontainers zu nutzen. Zusätzlich zu S3 brauchen wir allerdings auch den Lambda-Service von LocalStack, um ein Lambda zu erstellen und aufzurufen. Dies definieren wir entsprechend im Container für LocalStack. Zudem muss ein Netzwerk aufgesetzt werden, damit das Lambda mit S3 kommunizieren kann. Es wird nämlich automatisiert durch einen entsprechenden Request in der Methode *createLambdaFunktion()* erstellt. Dazu muss unsere Anwendung vorher als jar-Datei gepackt sein. Das nötigt uns, ein bisschen in die Maven-Trickkiste zu greifen, um den Test erst auszuführen, nachdem die Anwendung gepackt wurde. Mit dem failsafe-Plug-in ist es möglich, Tests mit der Endung „IT" nach der entsprechenden Stage in Maven auszuführen.

Der Ablauf des Tests ist anschließend sehr simpel. Es wird ein Bild nach S3 hochgeladen, das Lambda mit Referenz zu dem Bild angestoßen und der Rückgabewert anschließend auf seine Größe kontrolliert.

Das Erstellen des Lambdas ist hier der Übersicht halber aus *Listing 6* entfernt worden, soll aber abschließend noch erläutert werden. Mittels des AWS-SDK wird ein Request zum Erstellen des Lambdas an LocalStack abgeschickt *(siehe Listing 7)*. In diesem muss angegeben werden, welcher *EventHandler* ausgeführt werden soll. Zudem muss die Anwendung in Form einer jar-Datei übermittelt werden. Eine Reihe von Umgebungsvariablen dient dazu, dass das Lambda sich auch mit LocalStack und nicht mit der AWS-Cloud verbindet.

Zusammenfassung

Wir haben gesehen, wie man den Code innerhalb eines AWS-Lambdas strukturieren kann, um bessere Testbarkeit zu ermöglichen. Dependency Injection hilft uns dabei, in Isolation zu testen. Vor allem Unit-Tests werden so in ihrem Aufbau deutlich simpler.

Anhand eines Integration-Tests konnten wir sehen, wie die Integration zu AWS-spezifischen Komponenten durch LocalStack und Testcontainers testbar wird. Wir können so zum Beispiel verifizieren, dass der Code zum Herunterladen von Objekten von S3 auch wirklich funktioniert. Das alles lokal, ohne ein Deployment in die Cloud durchzuführen.

Der Component-Test ist im Aufbau ein wenig komplizierter. Er ermöglicht es allerdings zu verifizieren, dass beispielsweise die selbst erstellten Events auch funktionieren. Zudem hilft es bei Updates des AWS-Frameworks. Der Test gibt einem die Sicherheit, dass das Lambda auch nach Updates fehlerfrei läuft. Das alles ebenfalls lokal, ohne ein Deployment in die Cloud.

Der Code wurde, um eine bessere Lesbarkeit zu ermöglichen, ein wenig vereinfacht und reduziert. Der gesamte Code, inklusive Tests, ist ebenfalls auf GitHub *[4]* verfügbar.

Quellen

[1] *https://martinfowler.com/articles/microservice-testing/*
[2] *https://alistair.cockburn.us/hexagonal-architecture/*
[3] *https://localstack.cloud*
[4] *https://github.com/JensKnipper/testing-aws-lambdas*

Jens Knipper
OpenValue Düsseldorf GmbH
jens@openvalue.de

Jens Knipper ist Software Engineer bei OpenValue in Düsseldorf.

Er arbeitet gerne in verschiedenen Umgebungen, Architekturen und mit unterschiedlichen Technologien. Die Erkenntnisse, die er dadurch gewinnt, versucht er durch Vorträge und Blog-Artikel mit anderen zu teilen.

iJUG Verbund
www.ijug.eu

FÜR 29,00 €
BESTELLEN

Java aktuell

JAHRESABO

Mehr Informationen zum Magazin und Abo unter:
www.ijug.eu/de/java-aktuell