Serverlose Microservices

AWS Lambda

Michael Vitz

Seit Kurzem gibt es von verschiedenen Cloud-Anbietern Services für die Entwicklung serverloser Event-getriebener Anwendungen in der Cloud: Google (Cloud Functions), IBM (OpenWhisk), Microsoft (Azure Functions) und allen voran AWS Lambda von Amazon. Dieser Artikel beschreibt am Beispiel von AWS Lambda das Programmiermodell und hilft bei der Beantwortung der Frage, ob sich der Einsatz für Ihre Anwendung lohnt.

Was ist AWS Lambda?

Im Grunde bietet AWS Lambda dem Entwickler lediglich die Möglichkeit, eigenen Code in Form von Funktionen in der AWS-Cloud zu deployen und mit diesen auf Events innerhalb der Cloud zu reagieren. Die Funktionen können hierbei in JavaScript, Java [LambdaJava] oder Python [LambdaPython] entwickelt werden.

Events kommen von anderen AWS-Services und werden entweder zu AWS Lambda gepusht oder transparent von AWS Lambda durch Polling geholt. Für den Aufruf der Lambda-Funktion ist das Modell dabei zwar vollkommen transparent, für die Entwicklung macht dies jedoch einen kleinen, aber bedeutenden Unterschied. Wird die Funktion per Push aufgerufen, so muss man dem pushenden Service das Recht zuweisen, Lambda-Funktionen aufrufen zu dürfen. Werden Events gepollt, so benötigt Lambda das Recht, den Service abzufragen. Einen Überblick über die aktuell unterstützten AWS-Services und deren Modell bietet Tabelle 1.

Neben diesen Eventquellen kann man mit den AWS SDKs auch programmatisch eigene Events erzeugen, auf die eine Lambda-Funktion reagieren kann.

Die erste eigene Lambda-Funktion

AWS Lambda bietet dem Java-Entwickler zwei Möglichkeiten, um eine Lambda-Funktion zu implementieren. Die erste Möglichkeit besteht darin, eine Java-Funktion mit der in Listing 1 gezeigten Syntax zu schreiben.

```
OutputType handler-name(InputType input, Context context) {
  ...
}
```

Listing 1: Generische Lambda-Funktionssyntax

Alternativ kann man `com.amazonaws.services.lambda.runtime.RequestHandler` (s. Listing 2) oder `com.amazonaws.services.lambda.runtime.RequestStreamHandler` (s. Listing 3) aus der `aws-lambda-java-core`-Bibliothek implementieren. Beide Interfaces geben hierbei eine Methode mit obiger Syntax vor.

```
package com.amazonaws.services.lambda.runtime;
import com.amazonaws.services.lambda.runtime.Context;

/**
*
* Lambda request handlers implement AWS Lambda Function application
* logic using plain old java objects as input and output.
*
* @param <I> The input parameter type
* @param <O> The output parameter type
*/
public interface RequestHandler<I, O> {
  /**
  * Handles a Lambda Function request
  * @param input The Lambda Function input
  * @param context The Lambda execution environment context object.
  * @return The Lambda Function output
  */
  public O handleRequest(I input, Context context);
}
```

Listing 2: com.amazonaws.services.lambda.runtime.RequestHandler

```
package com.amazonaws.services.lambda.runtime;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;

/**
* Low-level request-handling interface, Lambda stream request handlers
* implement AWS Lambda Function application logic using input and
* output stream
*/
public interface RequestStreamHandler {
  /**
  * Handles a Lambda Function request
  * @param input The Lambda Function input stream
  * @param output The Lambda function output stream
  * @param context The Lambda execution environment context object.
  * @throws IOException
  */
  public void handleRequest(InputStream input, OutputStream output,
    Context context) throws IOException;
}
```

Listing 3: com.amazonaws.services.lambda.runtime.RequestStreamHandler

Die Wahl von Input- und Output-Typ hängt von den Events ab, die unsere Lambda-Funktion verarbeiten soll. Möglich sind die folgenden Typen:

- primitive Java-Typen (z. B. `String`)
- Stream-Typen

AWS-Service	Modell	Erzeugt ein Event, wenn
Simple Storage Service (S3)	Push	ein Objekt in einem Bucket angelegt oder gelöscht wird
DynamoDB	Pull	eine Änderung im DynamoDB Stream auftaucht
Kinesis	Pull	im Stream ein Ereignis auftaucht
Simple Notification Service (SNS)	Push	eine Nachricht zum passenden Topic gesendet wurde
Simple Email Service (SES)	Push	eine E-Mail eingetroffen ist
Cognito	Push	User-Daten oder Einstellungen geändert wurden
CloudWatch Logs	Push	ein Logeintrag stattfindet
CloudFormation	Push	die Lambda-Funktion aufgerufen wird
CloudWatch Events	Push	sich eine AWS Ressource ändert
Scheduled Events (über CloudWatch Events)	Push	der definierte Zeitpunkt eintritt

Tabelle 1: AWS-Services mit Lambda-Unterstützung

- vordefinierte AWS Event-Typen (z. B. `S3Event`) aus der `aws-lambda-java-events`-Bibliothek
- eigene POJOs

Die beiden letzten Typen werden dabei von Lambda mittels Serialisierung von und zu JSON genutzt.

Als Return-Wert kann auch `void` verwendet werden, wenn die Funktion asynchron aufgerufen wird. Ob eine Funktion synchron oder asynchron aufgerufen wird, hängt damit zusammen, welcher Service das Event erzeugt.

Der zweite Parameter vom Typ `com.amazonaws.services.lambda.runtime.Context` ist optional und kann deshalb im Falle der Nicht-Interface-Implementierung weggelassen werden. Dieser bietet dem Entwickler Zugriff auf den Ausführungskontext der Funktion, wie der Größe des der Funktion zugewiesenen Arbeitsspeichers oder der noch verfügbaren Zeit, bevor die Funktion in den definierten Timeout läuft.

In unserem Beispiel (s. Listing 4) haben wir uns für das Implementieren des Interfaces entschieden und nutzen eigene POJOs als In- und Output-Typ.

```
package com.innoq.aws.lambda.greet;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public final class GreetFunction implements
RequestHandler<GreetFunction.GreetRequest,
     GreetFunction.GreetResponse> {
  @Override
  public GreetResponse handleRequest(GreetRequest input,
      Context context) {
    return new GreetResponse("Hello, " + input.name + "!");
  }
  public static final class GreetRequest {
    private String name;
    public void setName(String name) { this.name = name; }
  }
  public static final class GreetResponse {
    private String text;
    public GreetResponse(String text) { this.text = text; }
    public String getText() { return text; }
  }
}
```

Listing 4: Eine einfache Java Lambda-Funktion

Deployment

Die in Java geschriebene Funktion wird (im Gegensatz zu den in JavaScript oder Python geschriebenen Lambda-Funktionen) nicht im Quellcode deployt, sondern als JAR-Datei. Hierzu können die im Java-Umfeld üblichen Build-Tools verwendet werden. Zu beachten ist lediglich, dass alle Abhängigkeiten mit eingepackt werden müssen. Dies gilt auch für die von Amazon bereitgestellten Lambda-Bibliotheken. Maven-Nutzer können hierzu zum Beispiel das `maven-shade-plugin` nutzen (s. Listing 5).

```
...
<plugin>
  <artifactId>maven-shade-plugin</artifactId>
  <version>2.3</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
    </execution>
  </executions>
</plugin>
...
```

Listing 5: maven-shade-plugin

Anschließend muss das so gebaute JAR natürlich noch deployt werden. Dies kann man per AWS Console im Browser oder über das AWS CLI erledigen. Da das CLI recht komplex ist, nutzen wir für unsere erste Funktion die Console im Browser (s. Abb. 1). Anschließend kann die Funktion getestet werden (s. Abb. 2 und 3).

Abb. 1: Funktion anlegen in der AWS Console

Abb. 2: Erzeugen eines Testevents in der AWS Console

Anbinden an eine Eventquelle

Als letzten Schritt muss man die Funktion an die gewünschte Eventquelle anbinden. Dieser Vorgang variiert je nach Quelle

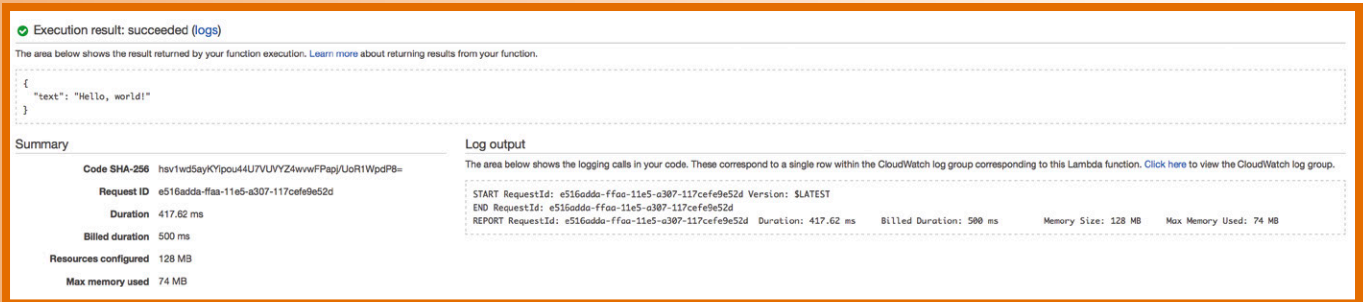

Abb. 3: Ergebnis eines Testlaufs mit dem Testevent in der AWS Console

und kann mitunter sehr komplex werden. Die Abbildungen 4 und 5 zeigen eine beispielhafte Integration der Lambda-Funktion in das API Gateway sowie einen Testaufruf.

interessiert nicht mehr, auf welchem Server seine Applikation läuft. Er definiert lediglich, welche und wie viele Ressourcen er braucht und welches Betriebssystem er gerne hätte (Infrastructure as a Service).

Schnell merkte man jedoch, dass man noch weitere abstrahieren kann. Für viele Anwendungen spielt auch das konkrete Betriebssystem keine Rolle mehr, sondern relevant ist lediglich, dass eine für den Code passende Ausführungsumgebung vorhanden ist. Im Falle der JVM ist dies mindestens eine passende Java-Runtime und gegebenenfalls ein Applikationsserver. Benötigt man weitere Infrastruktur, wie eine Datenbank oder eine Queue, so bietet einem die Cloud auch hierfür fertige Abstraktionen an.

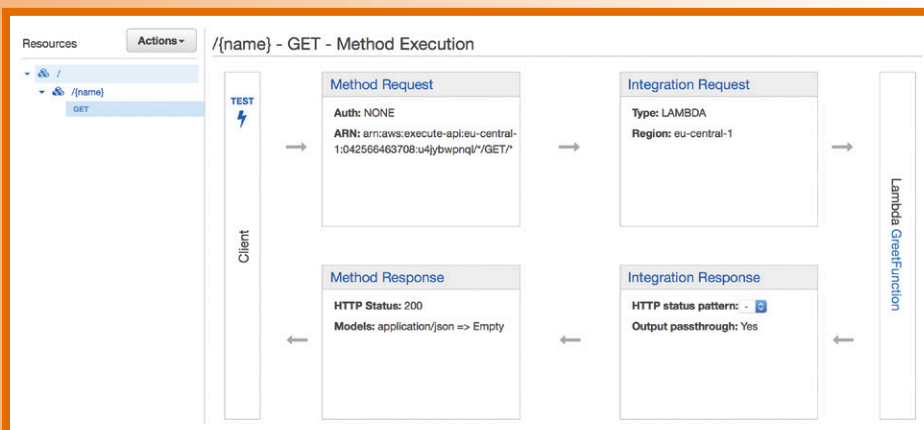

Abb. 4: Verbindung von API Gateway und Lambda

AWS Lambda geht an dieser Stelle noch einen Schritt weiter. Entwickler einer Lambda-Funktion sind vollständig von der Infrastruktur abgeschirmt. Die Cloud-Plattform kümmert sich selbstständig um Skalierung und um Monitoring. In wie vielen parallelen Instanzen ein Lambda-Code läuft, ist ebenfalls nicht selbst definierbar. Lediglich der für die Ausführung benötigte Arbeitsspeicher lässt sich definieren. Für diesen Ansatz wird der Begriff der „Serverless Application“ [ServerlessApps] genutzt. Dieser macht deutlich, dass es für den Entwickler so aussieht, als würde man keinen Server mehr brauchen, obwohl natürlich weiterhin ein Server im Hintergrund genutzt wird.

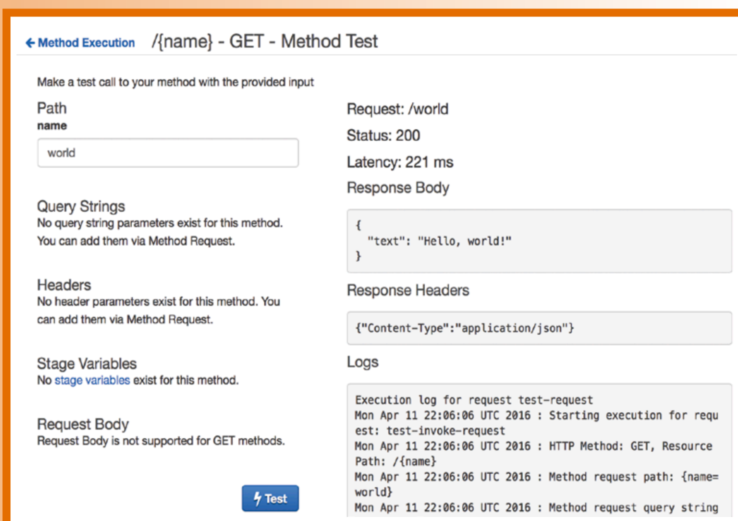

Abb. 5: Testaufruf des API Gateways

Vorteile

Die Vorteile von AWS Lambda liegen ganz klar in der bereitgestellten Abstraktion und im Preismodell. Nutzt man Lambda, so muss man sich um Infrastruktur wenig Gedanken machen und kann sich somit komplett auf die zu implementierende Fachlichkeit konzentrieren.

Die Kosten für AWS Lambda setzten sich hierbei aus Anforderungs- und Verarbeitungsgebühren zusammen. Die Anforderungsgebühren berechnen sich einfach durch die Anzahl der aufgerufenen Funktionen pro Monat. Die ersten 1.000.000 Aufrufe pro Monat sind dabei kostenlos. Anschließend zahlt man 0,20 $ für jede weiteren 1.000.000 Aufrufe.

Serverlose Applikationen

Mit Lambda hat es Amazon geschafft, die natürliche nächste Stufe einer Cloud-Abstraktion zu erschaffen.

Der erste Schritt der Cloud bestand darin, eine Abstraktion über physische Hardware zu ziehen. Den Nutzer einer Cloud

Zu den Anforderungs- kommen noch die Verarbeitungsgebühren, die sich aus der Zeit, die in einer Funktion verbraucht wird, und der Menge des verfügbaren Arbeitsspeichers zusammensetzen. Für 1 GB/s zahlt man hier aktuell 0,00001667 $, wobei die Zeit auf 100 ms gerundet wird. Auch hier gibt es ein kostenloses Kontingent pro Monat von 400.000 GB/s.

Hat man beispielsweise eine Funktion geschrieben, die auf bei _S3_ hochgeladene Dateien reagiert, und es werden pro Monat 5.000.000 Dateien hochgeladen, dann entstehen bei einer angenommenen durchschnittlichen Laufzeit von 500 ms und 512 MB zugewiesenem Arbeitsspeicher folgende Kosten:

- Anforderungsgebühren = 5.000.000 - 1.000.000 * 0,20 $ = 0,80 $
- Verarbeitungsgebühren = 5.000.000 0,5 GB * 0,5s - 400.000 GB/s * 0,00001667 $ = 14,17 $
- Gesamtkosten = 0,80 $ + 14,17 $ = 14,97 $

Nachteile

Neben den oben genannten Vorteilen gibt es natürlich auch Nachteile. Die vorhandene Dokumentation von Lambda ist gut und vollständig. Da zu einer Lambda-Funktion jedoch auch immer die Integration mit einer Eventquelle gehört, werden gerade AWS Neulinge aufgrund der vielen AWS spezifischen Begriffe und dem gleichzeitigen Lernen von mehreren AWS-Diensten geradezu erschlagen.

Unabhängig von der Lernkurve begibt man sich durch die Nutzung von AWS Lambda in eine starke Abhängigkeit zu Amazon. Ein Wechsel des Cloud-Angebots ist anschließend nur noch schwer möglich beziehungsweise zieht einen noch höheren Migrationsaufwand, als sowieso schon, mit sich. Je nach Verwendung von Lambda müsste die andere Cloud dieselben Events zur Verfügung stellen. Selbst wenn dies der Fall ist, muss man davon ausgehen, dass diese im Detail unterschiedlich sind und man deshalb seine Funktionen anpassen muss.

Fazit

Die Cloud-Services wie AWS Lambda heben den Level des Cloud-Gedankens. Dass dahinter ein funktionierendes Modell steht, zeigt sich nicht zuletzt daran, dass die Konkurrenten wie Google oder Microsoft mittlerweile darauf reagiert haben und ihre eigenen Konkurrenzprodukte anbieten. Für uns als Entwickler heißt das, dass wir eine integrierte Technologie an die Hand bekommen, mit der wir elegant Probleme lösen können, Datenflüsse oder Ereignisbehandlungen zu realisieren. Auch wenn Lambda natürlich keine Lösung für alles ist, lohnt es sich trotzdem, den Ansatz von serverlosen Anwendungen weiter zu beobachten. Die Möglichkeit, eine Public Cloud zu nutzen, nimmt immer weiter zu, insbesondere seitdem die Cloud-Anbieter anfangen, auch Regionen ihrer Cloud in Deutschland aufzubauen. Zudem werden auch Cloud-Produkte für Private Clouds in Zukunft AWS Lambda ähnliche Services zur Nutzung anbieten.

Aktuell empfiehlt sich deswegen der Einsatz einer AWS Lambda-Architektur für Anwendungen, die sowieso in der Cloud deployt werden sollen und die sehr stark Event-getrieben arbeiten. In Verbindung mit dem API Gateway sind jedoch auch komplette auf Lambda basierende Webanwendungen denkbar. Die dadurch gewonnene Freiheit von Infrastruktur und den attraktiven Preis zahlt man allerdings mit der stärkeren Abhängigkeit zu AWS.

Links

[AzureFunctions] N. Mashakowski, 31.3.2016, https://azure.microsoft.com/en-us/blog/introducing-azure-functions/

[GoogleCloudFunctions] J. Novet, 9.2.2016, http://venturebeat.com/2016/02/09/google-has-quietly-launched-its-answer-to-aws-lambda/

[IBMOpenWhisk] M. Behrendt, 22.2.2016, https://developer.ibm.com/openwhisk/2016/02/22/announcing-ibm-bluemix-openwhisk/

[LambdaAnnouncement] J. Barr, AWS Lambda – Run Code in the Cloud, 13.11.2014, https://aws.amazon.com/blogs/aws/run-code-cloud/

[LambdaJava] J. Barr, 15.6.2016, https://aws.amazon.com/de/blogs/aws/aws-lambda-update-run-java-code-in-response-to-events/

[LambdaPython] J. Barr, 8.10.2015, https://aws.amazon.com/de/blogs/aws/aws-lambda-update-python-vpc-increased-function-duration-scheduling-and-more/

[ServerlessApps] K. Fromm, 15.10.2015, http://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless/

Michael Vitz ist Consultant bei innoQ und verfügt über mehrjährige Erfahrung in der Entwicklung und im Betrieb von JVM-basierten Systemen. Zurzeit beschäftigt er sich vor allem mit den Themen DevOps, Continuous Delivery und Cloud-Architekturen sowie Clojure.
E-Mail: michael.vitz@innoq.com

Staffelwechsel

Manchmal ist es Zeit für eine Erneuerung. In den letzten Jahren durfte ich mit vielen tollen Koautoren zu unterschiedlichsten Themen Beiträge in dieser Kolumne schreiben.

Ich bin dankbar für die umfassende Unterstützung durch die JavaSPEKTRUM-Redaktion. Meine Koautoren und ich konnten uns stets voll und ganz auf das tolle Redaktionsteam verlassen.

Für mich kommt jetzt die Zeit, intensiver einige der hier vorgestellten Technologien anzuwenden. Nun ist es soweit, meinen Nachfolger zu begrüßen. Ich freue mich, dass nun mein geschätzter Kollege Michael Vitz den Praktiker fortführt. Michael, ich freue mich auf den frischen Wind, den Du unserer Kolumne bringen wirst. Möge die Macht mit Dir sein!

Sowohl den Lesern als auch der Redaktion wünsche ich tolle Artikel, einen frischen Wind und inspirierende Themen.

Phillip Ghadir im April 2016