Abkehr vom monolithischen Entwickeln

Tröpfchenweise

Constantin Söldner

In der Cloud bieten sich Entwicklern neue Wege, Anwendungen ressourcensparend und effizient zu hosten. Serverless Programming hat das Potenzial, völlig neue Softwarearchitekturen zu etablieren, bei denen Anbieter von Plattformdiensten das Ausführen des Codes sowie Aspekte wie Skalierbarkeit oder Ausfallsicherheit sicherstellen.

Zum Entwickeln und Betreiben von Applikationen in der Cloud bietet Serverless Programming entscheidende Vorteile im Vergleich zum klassischen monolithischen Ansatz. Es befreit Entwickler von administrativen Aufgaben, wie dem Konfigurieren und Warten von Applikationsservern, dem Bereitstellen ausreichender Ressourcen im Rahmen der Skalierbarkeit oder dem Garantieren der Ausfallsicherheit.

Serverless Programming verzichtet allerdings keineswegs auf einen Applikationsserver. Vielmehr stellen Cloud-Anbieter eine Laufzeitumgebung bereit, die sie hosten und betreiben. Der Entwickler muss somit dazu seinen Code lediglich hochladen. Der Platform-as-a-Service-Anbieter (PaaS) sorgt für das Ausführen des Codes und kümmert sich etwa um Skalierbarkeit und Ausfallsicherheit. Der Entwickler stellt den Programmcode im Normalfall nicht als komplette Anwendung bereit, sondern in Form von Ausführungseinheiten. Die Aufteilung von Funktionen in Services kann als Gegenmodell zu monolithischen Anwendungen verstanden werden. Daher spricht man in diesem Zusammenhang auch von Microservices.

Klein, aber fein

Im Gegensatz zum klassischen Cloud-Computing müssen Entwickler beim Serverless Programming keine eigenen virtuellen Maschinen verwalten. Dies befreit sie vor allem von administrativen Aufgaben. Serverless Programming bietet zudem finanzielle Vorteile, weil Anbieter solcher Plattformen in der Regel verbrauchsabhängig abrechnen; der Kunde zahlt nur für das Ausführen seines Codes; wird der Software-Code nicht ausgeführt, fallen keine Gebühren an. Auf der anderen Seite birgt der Einsatz von Microservices neue Herausforderungen. So verbietet sich das in der Vergangenheit praktizierte monolithische Entwickeln von Anwendungen bei Microservices. Es eignet sich daher eher für von Grund auf neu zu entwickelnde Anwendungen.

Verschiedene Cloud-Anbieter ermöglichen das Bereitstellen von Code als Microservices. Neben AWS Lambda sind besonders Iron.io, Microsoft Azure Functions, Google Cloud Functions und Bluemix OpenWhisk (IBM) erwähnenswert. Allerdings sind Azure Functions und Google Cloud Functions recht neu und haben noch Preview-Status. Am längsten im Geschäft ist iron.io, dem bereits zahlreiche größere Kunden vertrauen. Für AWS Lambda spricht vor allem die gute Integration in das AWS-Ökosystem. Weitere Einzelheiten zu deren übrigen Anbietern finden sich im Kasten „Alternativen zu AWS Lambda“.

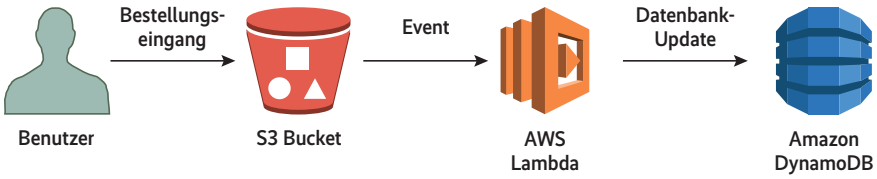
Lambda unter der Lupe

AWS Lambda integriert weitere Dienste von Amazon, etwa solche zur Datenspeicherung (S3, DynamoDB), Messaging-Services (SQS), Security-Dienste oder Services zum Generieren von Events. Dadurch lassen sich komplexere Anwendungen auf Basis von Microservices aufbauen. Die enge Verflechtung der AWS-Dienste ermöglicht eine ganze Reihe interessanter Einsatzszenarien.

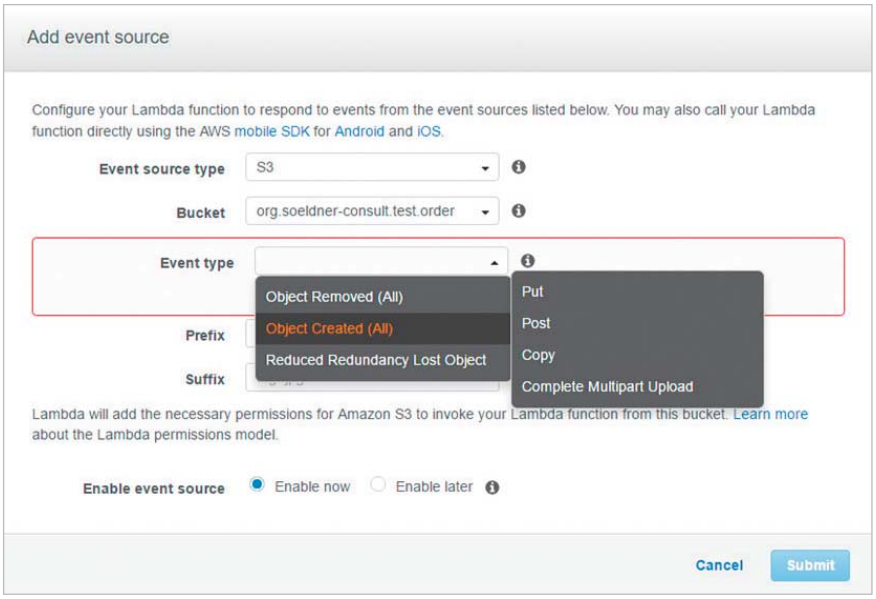
Häufig anzutreffen ist das Event-basierte Ausführen von Lambda-Funktionen, etwa beim Ablegen einer neuen Datei im Object Store S3. Damit lässt sich beispielsweise das Verarbeiten strukturierter Dateien im JSON- oder XML-Format automatisieren. Um typische Beispiele handelt es sich beim Verarbeiten von Log-Dateien oder das Prozessieren von Produktionsdaten. Quellen für solche Events können weitere Amazon-AWS-Dienste wie DynamoDB, Kinesis, SNS, SES oder Cognito sein. Damit eignet sich der Dienst außerdem für die Verarbeitung großer Datenmengen in Echtzeit, was besonders bei Big-Data-Szenarien erforderlich ist.

Lambda-Funktionen müssen nicht notwendigerweise durch Events ausgelöst werden. Entwickler können sie mit API-Gateways als Webservice publizieren und so für andere Anwendungen (innerhalb und außerhalb der eigenen Amazon-Umgebung) zur Verfügung stellen. Dieser Ansatz bietet sich insbesondere für das Umsetzen größerer Anwendungen als Microservices an.

Überdies lässt sich Lambda im Rahmen der Infrastrukturautomatisierung verwenden. Amazon bietet hierfür den CloudFormation-Dienst. Dieser erlaubt es, ganze Umgebungen (virtuelle Maschinen, Netzwerke, Datenbanksysteme) per Knopfdruck anzulegen. Allerdings ermöglicht CloudFormation keine direkte Ausführung von prozeduralem Code. AWS Lambda lässt sich in diesem Zusammenhang zum Durchführen komplexer Konfigurationen einsetzen, die nicht mehr mit CloudFormation abgebildet werden können, sondern die in einem eigenen Programm gelöst werden müssen.



AWS Lambda im Zusammenspiel mit S3 und DynamoDB (Abb. 1)



Erstellung eines Events in S3 (Abb. 2)

Sorgfältige Planung beim Lambda-Entwurf

Vor dem Einsetzen einer Lambda-Funktion muss der Entwickler die Region festlegen, in der Amazon sie hostet. Das Unternehmen teilt seine globalen Rechenzentren in Regionen auf, um neben hoher Verfügbarkeit geringe Latenzzeiten für den Kunden zu gewährleisten. Allerdings bietet es neuere Dienste häufig nicht in allen Regionen an. Zum aktuellen Zeitpunkt gibt es AWS Lambda in den Regionen US East (N. Virginia), US West (Oregon), EU (Irland), EU (Frankfurt) und Asia Pacific (Tokyo).

Außerdem muss der Entwickler beim Entwerfen einer Lambda-Funktion bestimmte Aspekte berücksichtigen: Er muss den Ressourcenbedarf der Anwendung abschätzen und konfigurieren. Das ist insbesondere aus Kostensicht relevant. Zudem muss er gewisse Ressourcenlimits beachten. Eine Lambda-Funktion darf eine maximale Laufzeit von 300 Sekunden aufweisen, andernfalls wird ein „exceeded limits“-Fehler zurückgegeben. Zudem ist der temporäre Festplattenspeicher auf 512 MByte pro Funktionsaufruf begrenzt. Die gezippte Funktion darf inklusive ihrer Abhängigkeiten 50 MByte nicht überschreiten. An Sprachen versteht Lambda zum aktuellen Zeitpunkt Node.js, Python sowie Java.

Hinsichtlich des Verbrauchs während des Funktionsablaufs muss der Entwickler lediglich den benötigten Speicher abschätzen. Er kann den allozierten Speicher zwischen 128 MByte und 1536 MByte RAM in Schritten von 64 MByte festlegen. Aus Effizienzgründen sollte er daher den Speicherbedarf seiner Funktion testen und anpassen. Wird ihr zu wenig Speicher zugeteilt, bricht sie mit einer Fehlermeldung ab. Bei höherer Zuteilung entstehen im Gegenzug höhere Kosten.

Konfiguration (und Verwaltung)

Neben dem eigentlichen Bereitstellen der Lambda-Funktion gilt es, einige Konfigurationsaspekte zu beachten, was sich anhand eines Auftragsverarbeitungssystems für eingehende Kundenaufträge skizzieren lässt (Abbildung 1):

1. Der Kunde setzt über ein Portal einen Auftrag ab. Dieser wird als JSON-Datei in AWS S3 abgelegt.
2. Die Ablage der json-Datei in S3 generiert ein Event, das wiederum eine AWS-Lambda-Funktion auslöst.
3. Die Lambda-Funktion liest das JSON-Dokument aus und speichert die zu verarbeiteten Daten in eine DynamoDB-Datenbank.

Das Anlegen der neuen Bestellung kann über eine Kundenwebseite erfolgen. Eventuell hat der Kunde direkten Zugriff auf den S3 Bucket. In diesem Beispiel besteht die Bestellung aus einem JSON-Dokument. Andere Formate wie XML sind ebenfalls denkbar. Damit

iX-TRACT

- Microservices stehen im Gegensatz zu monolithischen Anwendungen und sollten als alternative Architektur bei Neuentwicklungen in Betracht gezogen werden.
- Sie skalieren sowohl in Leistung als auch im Preis feinkörniger als virtuelle Maschinen.
- AWS Lambda bietet die Integration in ein umfassendes Ökosystem, erlaubt jedoch keine Installation in der Private Cloud.

Listing der Lambda-Funktion

```
console.log('Loading function');

var AWS = require('aws-sdk');
var doc = require('dynamodb-doc');
var dynamo = new doc.DynamoDB();
var s3 = new AWS.S3();

exports.handler = function(event, context) {
    console.log('Received event:',
                JSON.stringify(event, null, 2));

    var params = {Bucket:
    event.Records[0].s3.bucket.name, Key:
    event.Records[0].s3.object.key};
    var dynamoEvent = {};

    s3.getObject(params, function(err, data) {
        if (err)
            console.log(err, err.stack);
        else {
            console.log(data);

            var dynamoData = data.Body.toString('ascii');
            var uid = event.Records[0].s3.object.key;

            dynamoEvent = {
                TableName: "order",
                Item: { id: uid, text: dynamoData }
            };
            dynamo.putItem(dynamoEvent, context.done);
        }
    });
};
```

die Ablage einer neuen Bestellung die richtige Lambda-Funktion auslöst, ist für den S3 Bucket ein Event zu konfigurieren (Abbildung 2).

Sobald ein Benutzer eine Bestellung in den S3 Bucket abgelegt hat, wird die AWS-Lambda-Funktion ausgeführt. S3 verwendet dafür das Push Event Model: die auszuführende Lambda-Funktion wird in S3 spezifiziert. Das Listing zeigt eine Implementierung mit Node.js, die die Bestellung im JSON-Format verarbeitet, die relevanten Informationen extrahiert und eine Dynamo-DB-Tabelle updatet.

Bei DynamoDB handelt es sich um einen von Amazon verwalteten Dienst, mit dem sich NoSQL-Datenbanken anlegen lassen. Statt DynamoDB lassen sich auch andere Datenbanksysteme anbinden. Mit dem Aufrufen von *console.log ('Loading function')* kann der Entwickler in die Log-Datei für Lambda schreiben, die wiederum in Amazons zentralen Logging-Dienst CloudWatch abgelegt wird. Die Funktion *require* importiert die benötigten Node.js-Module (in diesem Fall *aws-sdk* und *dynamodb-doc*). Der Aufruf *exports.handler = function(event,context)* gibt die Funktion an, die das generierte Ereignis verarbeitet. Es wird dabei als JSON-Object übergeben, über das AWS Lambda Zugriff auf die Metadaten (wie Objektname oder Name des S3 Buckets) des neuen Kundenauftrags hat. Mit diesen Metadaten erfolgt der Zugriff auf das eigentliche Objekt. Die Funktion *getObject* liest den Inhalt der in der Variable *S3* referenzierten Datei ein und legt deren Inhalt in der Variablen *data* ab. *data.body.toString('ascii')* interpretiert den gespeicherten Inhalt als Ascii-String und speichert ihn im letzten Schritt mit der Methode *dynamo.putItem* in die Tabelle *Order*. Damit AWS Lambda Zugriff auf DynamoDB hat, müssen ausreichende Berechtigungen existieren. Der Entwickler muss dazu eine Rolle im zentralen Berechtigungsmanagement-Dienst IAM erstellen, die er der Lambda-Funktion zuordnet.

Monitoring von Lambda-Funktionen

Lambda bietet kein integriertes Debugging. Der Entwickler muss seinen Code lokal debuggen, bevor er ihn in AWS Lambda bereitstellt. Alternativ kann er ein Plug-in für Eclipse oder Visual Studio verwenden, um die Lambda-Funktion von dort aus auf Fehler zu überprüfen. Für das Monitoring vorhandener Lambda-Funktionen ist der Entwickler daher hauptsächlich auf Logs angewiesen. Zum Auswerten der Log-Dateien kann er AWS CloudWatch

Alternativen zu AWS Lambda

Iron.io: Der Dienst wurde bereits 2010 gegründet und ist damit einer der erfahrensten Player im Serverless-Programming-Markt. Technisch setzt Iron.io im Hintergrund auf Docker. Neben der Cloud-Variante unterstützt es die On-Premise-Ausführung. Positiv hervorzuheben ist auch die breite Sprachunterstützung. Iron.io unterstützt Ruby, Python, PHP, Java, .NET, Node.js und Go; des Weiteren eine maximale Laufzeit von einer Stunde pro Funktionsaufruf. Das Limit für die Speichernutzung pro Funktion beträgt 2048 MByte. Daneben soll das Projekt Kratos zukünftig auch das Ausführen von Lambda-Code unterstützen. Aufgrund seiner mehrjährigen Erfahrung und seinen namhaften Kunden wie Google oder Twitter ist Iron.io einer der Hauptakteure im Serverless Computing Markt.

Azure Functions (Microsoft): Azure Functions ist Microsofts Antwort auf AWS Lambda und lag zum aktuellen Zeitpunkt lediglich in einer Preview-Version vor. Ähnlich wie AWS Lambda erlaubt Azure Functions das event-basierte Ausführen von Code (etwa über Azure Storage Blob Containers oder Azure Queues). Interessant ist auch, dass Microsoft die Laufzeitumgebung von Azure Functions unter der MIT-Lizenz als Open Source veröffentlicht hat. Damit können Firmen ihre Microservices im heimischen Rechenzentrum betreiben. Zudem lassen sich Azure Functions über eine HTTP-basierte API betreiben. Als maximalen Speicherverbrauch pro Funktion können Nutzer 1536 MByte wählen. Ein Vorteil ist, dass die Laufzeit einer Funktion nicht limitiert ist.

Cloud Functions (Google): Der derzeit im Alpha-Release vorliegende Dienst Cloud Functions ist Googles Versuch, im Serverless-Programming-Markt Fuß zu fassen. Es versteht aktuell nur JavaScript in einer Node.js-Umgebung. Google bietet aber ähnlich wie AWS eine Integration in andere eigene Dienste an. So unterstützt der Dienst asynchrone Events von Google Cloud Storage und von Google Cloud Pub/Sub. Daneben lassen sich die Funktionen synchron per HTTP Call aufrufen. Allerdings hält sich Google mit Marketing für den neuen Dienst noch zurück. Interessenten müssen zudem ein Formular ausfüllen, um den Dienst nutzen zu können. Hinsichtlich der Laufzeit einer Funktion gibt es auch bei Cloud Functions keine Beschränkungen.

Bluemix OpenWhisk (IBM): OpenWhisk ist die Microservice-Lösung von IBM. Es ist Open Source und steht unter der Apache-2.0-Lizenz. Damit will IBM vor allem die Entwicklung des Ökosystems rund um OpenWhisk und dessen angrenzende Dienste fördern. Als Programmiersprachen unterstützt OpenWhisk JavaScript, JavaScript für Node.js sowie Swift. Die Anwendungen können aber auch Binärprogramme sein, die in Docker-Container gekapselt sind. Das Zeitlimit für eine Funktionsausführung beträgt 5 Minuten, der maximale Speicherverbrauch ist mit 512 MByte gedeckelt.

verwenden. Neben den Log-Dateien werden zahlreiche Metriken wie die Anzahl der Request oder die durchschnittliche Latenz gespeichert. CloudWatch beherrscht benutzerdefinierte Alarme. Diese können sowohl auf Metriken als auch auf Log-Dateien basieren und etwa automatisiert E-Mails verschicken. Der Benutzer kann konfigurieren, in welchen Fällen ein Alarm ausgelöst wird: falls die Lambda-Funktion eine gewisse Laufzeit überschreitet oder auch wenn mehrere Funktionsausführungen fehlgeschlagen sind.

Um einer Microservices-Architektur gerecht zu werden, muss sich die Möglichkeit bieten, Lambda-Funktionen als Webservices (bevorzugt RESTless) zu publizieren. Amazon bietet dazu das sogenannte API Gateway. Hierbei handelt es sich um einen AWS-Dienst, der das Bereitstellen und Warten von Webservices erlaubt. Das Erstellen ist einfach: Der Benutzer vergibt dazu nur einen API-Namen, erstellt eine Ressource (das macht einen Teil der URL aus) sowie eine Methode, in diesem Fall POST, die auf die Lambda-Funktion gemappt wird. Damit lässt sich die Lambda-Funktion etwa über die folgende URL aufrufen:

```
https://my-api-id.execute-api.
            region-id.amazonaws.com/prod/order
```

my-api-id ist eine von Amazon generierte eindeutige ID (wie *w0ycdtcd4f*). *region-id* gibt die Amazon-Region an, in der die Lambda-Funktion liegt. *prod* entspricht der Ressource und *order* ist der API-Name. Im letzten Schritt muss der Anwender noch Berechtigungen setzen, damit nur bestimmte Personen den neu erstellten Webservice aufrufen können. Der Webservice kann auch öffentlich sein. Das Berechtigungsmanagement erfolgt dabei über den AWS-IAM-Dienst.

Geschickte Planung minimiert die Kosten

Eines der Hauptargumente für den Einsatz von Serverless Programming sind die geringen Kosten, die im Public-Cloud-Betrieb auf den Verzicht auf eigene Hardware zurückzuführen sind. Trotzdem gibt es Szenarien, in denen die klassische Ausführung von Programmcode auf einem eigenen virtuellen Server günstiger ist. Der Entwickler sollte daher vor dem Einsatz von AWS Lambda die Nutzungsszenarien kostenseitig durchrechnen. Generell hängt der monatliche Preis von drei Faktoren ab: der Anzahl der Ausführungen, dem allozierten Speicher für das Ausführen der Lambda-Funktion sowie deren Ausführungszeit.

Amazon stellt einen sogenannten „Free Tier“ pro Monat bereit, im Rahmen dessen keine Gebühren entstehen. Gebühren verlangt Amazon erst ab einer Mindestanzahl von 1 000 000 Ausführungen und ab 400 000 GByte-Sekunden pro Monat. Ein Benutzer kann damit eine Lambda-Funktion mit einem maximalen Speicherverbrauch von 128 MByte und einer durchschnittlichen Laufzeit von 20 Sekunden insgesamt 160 000 mal ausführen, ohne dafür zahlen zu müssen. Damit lassen sich im Rahmen des Free Tiers schon einige Testwandungen sowie kleinere Produktiv-Anwendungen betreiben.

iX-Wertung

⊕ Integration in das große AWS-Ökosystem
⊕ großzügiger kostenloser Sockel

⊖ keine On-Premise-Variante
⊖ stark eingeschränkte Laufzeit der Funktionen

Fazit

Serverless Programming bieten durchaus Vorteile. Neben dem Potenzial zur Kostensenkung und Skalierbarkeit fällt der Administrationsaufwand für Applikationsserver faktisch weg. Ein weiteres entscheidendes Kriterium ist die Integrierbarkeit mit weiteren Diensten. Cloud-Anbieter wie Amazon AWS und Microsoft Azure haben dabei den Vorteil, dass sie eine Reihe von Diensten anbieten, die sich einfach mit Microservices koppeln lassen. So lassen sich Lambda-Funktionen leicht durch Events von anderen AWS-Diensten auslösen. Ein Nachteil besteht allerdings darin, dass sie (im Gegensatz zu den Äquivalenten anderer Anbieter) nicht On-Premise ausgeführt werden können. Diese Option ist besonders dann interessant, wenn Bedarf nach größerer Anpassung besteht. AWS Lambda ist in dieser Hinsicht noch sehr eingeschränkt. Zudem ist die die maximale Ausführungsdauer einer Lambda-Funktion auf 5 Minuten begrenzt. (jab)

Constantin Söldner

ist bei der Söldner Consult GmbH in Nürnberg für den Bereich Cloud Consulting (AWS, MS Azure, vRealize Automation) zuständig.

Alle Links: www.ix.de/ix1608078