Serverless mit Fn Project

Dr. Frank Munz, munz & more

Das vorherrschende Cloud-Thema im Jahr 2017 war „serverlose Architekturen“. So gab es auf der Devoxx in Belgien, einer der besten Entwickler-Konferenzen in Europa, gleich sieben verschiedene Präsentationen zu diesem Thema.

Dieser Artikel, eine Übersetzung des auf Oracle Developer im März publizierten Originalartikels von Dr. Frank Munz *[1]*, ist eine Einführung in Fn Project. Es unterscheidet sich von den meisten anderen serverlosen Lösungen: Fn ist Cloud-agnostisch, polyglott, Open Source und hat nur Docker als einzige Abhängigkeit. Es ist außerdem brandneu; Fn wurde auf der JavaOne 2017 als Open Source freigegeben.

Dieser Artikel behandelt folgende Punkte: Zunächst werden die Grundlagen geklärt und dem Entwickler eine kurze Einführung in die serverlose Welt gegeben. Dabei kommen die einzigartigen Vorteile zur Sprache, die oft nicht ganz eindeutig definiert sind. Zweitens, und am wichtigsten, wird der schnelle Einstieg in die Programmierung mit dem neuen Fn Project gezeigt. Für eine praxisnahe Entwicklung werden Go und Java, Monitoring, Tests, lokale Entwicklung, JSON-Parameterübergabe, Verwendung von Docker Hub und der Einsatz von Fn in der Cloud behandelt. Drittens gibt es eine Übersicht der aktuellen Ankündigungen und darüber, was als Nächstes zu erwarten ist. Fn ist erst der Anfang einer Reise in die serverlose Welt.

„Serverlos“ ist offensichtlich kein besonders kluger Name. Ganz ehrlich: Die IT-Branche ist schlecht bei der korrekten Namensgebung und präzisen Definition von neuen Konzepten. Cloud Computing läuft nicht irgendwo in den Wolken, Data Lakes sind nicht nass und serverlos braucht Server. Deshalb zunächst eine Definition von „serverlos“ und der damit verwandten Konzepte.

Function as a Service (FaaS)

Function as a Service (FaaS) startete als Cloud-Service im Jahr 2014 mit AWS Lambda. Die Idee dahinter war einfach

und revolutionär zugleich: Es wird Quellcode ausgeführt, aber man muss sich keine Gedanken über die darunter liegenden Laufzeit-Umgebungen der Programmiersprachen, die Container, die virtuelle Maschine oder die Server machen. Im einfachsten Falle kopiert der Entwickler seinen Quellcode, fügt ihn in eine Maske des FaaS-Service in der Cloud ein und der Code wird ausgeführt.

Technisch gesehen sind Cloud-basierte FaaS-Lösungen auf der Basis von Containern implementiert (also ähnlich wie Docker, aber nicht unbedingt mit Docker selbst). Dieser Container ist für den Endanwender üblicherweise nicht sichtbar. Eine serverlose Funktion wird nur ausgeführt, wenn sie durch ein Ereignis ausgelöst wird; deshalb heißt es auch „ephemeral compute“. Bei FaaS gibt es also keinen permanent laufenden Server für einen Benutzer; es existiert auch keine Laufzeit-Umgebung wie WebLogic oder Tomcat mit einer permanent geöffneten IP-Adresse und Port. Ereignisse, die eine solche serverlose Funktion auslösen können, hängen vom Cloud Provider ab. Übliche Beispiele für Ereignisquellen sind: ein Datei-Upload, eine REST-Anfrage oder eine Nachricht, die von einem Nachrichtensystem aufgenommen wird.

Was FaaS in Public Clouds besonders interessant macht, ist der Umstand, dass nur pro Aufruf der Funktion bezahlt werden muss. Außerdem ist die Skalierung automatisiert, eine Konfiguration für die Anzahl der Funktions-Instanzen ist also nicht erforderlich. Das Konzept „nicht zahlen für Leerlauf“ ist verlockend. Einige Anwendungsfälle berichten von Kosteneinsparungen von ein bis zwei Größenordnungen, wenn eine traditionelle, serverbasierte Anwendung ersetzt wurde [2].

Man sollte allerdings fair sein und auch die Kehrseite betrachten. Die wesentlichen Bedenken bei heutigen FaaS-Implementierungen betreffen die Abhängigkeit von einem bestimmten Cloud Anbieter, den sogenannten „Vendor Lock-in“:

- Die Funktion selbst wird über ein typisiertes, Cloud-Provider-spezifisches Ereignis ausgelöst.
- Die Verdrahtung verschiedener Ereignisquellen mit serverlosen Funktionen erzeugt eine harte Abhängigkeit zwischen der serverlosen Funktion und weiteren Diensten des Cloud-Providers.

Das Konzept von FaaS entwickelt sich jedoch weiter. So gibt es Diskussionen unter Software-Architekten, ob Funktionen besser gleich als Container implementiert werden sollten [3]. Derzeit legt jedoch keiner der großen Public-Cloud-Provider den Container offen, der intern für die Ausführung serverloser Funktionen verwendet wird.

Microservices vs. FaaS

Eine Microservices-Architektur versucht, eine Anwendung als eine Menge unabhängiger Services zu implementieren. Jeder Service läuft in seinem eigenen Prozess und hat seine eigenen Daten; untereinander kommunizieren die Services über ein leichtgewichtiges Protokoll [4]. Es stellt sich daher die Frage, ob FaaS nicht auch nur eine Art von Microservice ist. Kurz: FaaS erfüllt theoretisch die Definition eines Microservice. Allerdings müssen serverlose Funktionen in der Praxis erst zu einem Microservice zusammengefasst werden. Aber wie?

„Serverlos“ ist ein Architektur-Trend, der darauf ausgelegt ist, „alle Beziehungen zur Infrastruktur“ zu reduzieren [5]. FaaS ist also serverlos. Ein serverloser Cloud-Service ist ein PaaS mit echter Bezahlung nach Benutzung und automatisierter Skalierbarkeit. Als Beispiel für einen serverlosen Cloud-Service kann man sich einen Nachrichtendienst vorstellen. Wenn nur für die Anzahl der erzeugten und verarbeiteten Nachrichten bezahlt wird und der Service automatisch skaliert, können wir ihn berechtigterweise „serverlos“ nennen. Wenn Server mit den Message-Brokern sichtbar sind und pro Stunde für diese Server bezahlt wird, unabhängig davon, ob Nachrichten erzeugt oder verarbeitet werden, dann ist der Service nicht serverlos.

Derzeit gibt es mehr als ein Dutzend FaaS-Frameworks oder -Plattformen (eine Übersicht über die Projekte siehe [6]). Diese Projekte lassen sich je nach Zielrichtung in drei unterschiedliche Kategorien einteilen, wobei jede Kategorie typischerweise die Eigenschaften der vorherigen enthält [7]:

- *Komplexität*
  Die Komplexität einer Cloud-basierten FaaS-Implementierung eines bestimmten Anbieters wird reduziert, etwa durch die Konfiguration eines API-Gateways und Zugriffsverwaltung, die für eine REST-basierte Funktion benötigt wird. Ein typisches Beispiel für diese Kategorie ist AWS Chalice.
- *Portierbarkeit*
  Das Framework dient zur Abstraktion zwecks Portierbarkeit und zur zusätzlichen Vereinfachung der FaaS-Implementierung verschiedener Public-Cloud-Anbieter. Ein beliebtes Beispiel ist das „serverless.com“-Framework.
- *Standards*
  Es handelt sich um eine Standard-basierte, serverlose Plattform oder ein Framework, um laufende Funktionen vom Betrieb der Server zu abstrahieren. Diese Frameworks werden üblicherweise ohne einen Bezug zu einem bestimmten Cloud-Provider entwickelt. Wenn man solch ein Framework auf IaaS betreibt, werden Server abstrahiert, automatische Skalierung ist möglich, aber durch das IaaS-Preismodell wird keine echte Pro-Aufruf-Basis geschaffen. Beispiele für diese Kategorie sind Open FaaS und Fn Project.

Fn Project

Fn Project ist eine serverlose Plattform mit einer Vielzahl einzigartiger Vorteile: Es ist Container-basiert, polyglott, Cloud-agnostisch und hat Docker als einzige Abhängigkeit. Zum derzeitigen Zeitpunkt ist Fn Project eine Software-Plattform und es existiert noch kein FaaS-Cloud-Service. Die Entwicklung eines solchen Cloud-basierten Service wurde allerdings von Oracle angedeutet. Um alle Eigenschaften von Fn zu verstehen, ist es am einfachsten, Fn in Aktion zu erleben und selbst einige Funktionen laufen zu lassen.

Fn lässt sich leicht auf Windows- und Unix-Systemen mit dem einzeiligen Kommando „$ curl -LSs https://raw.githubusercontent.com/fnproject/cli/master/install | sh“ installieren. Auf Mac OS wird Fn mit „brew“ installiert. Einzelheiten zur Installation siehe [8].

Grundlagen mit Go

Um die Fn-Eigenschaften zu verstehen, zu Beginn eine einfache Go-Funktion.

```
# create oradev and with boilerplate for go
$ fn init --runtime go oradev
$ cd oradev
```

Listing 1

```
$ fn run

Building image oradev:0.0.1 ..
{"message":"Hello World"}
```

Listing 2

```
$ tree

.
├── func.go
├── func.yaml
└── test.json
```

Listing 3

```
$ fn deploy --app mygo --local

Deploying oradev to app: mygo at path: /oradev
Bumped to version 0.0.2
Building image oradev:0.0.2 ..
Updating route /oradev using image oradev:0.0.2...
```

Listing 4

```
# check deployed applications
$ fn apps list
mygo

# check existing routes
$ fn routes list mygo

path   image      endpoint
/oradev   oradev:0.0.2   localhost:8080/r/mygo/oradev
```

Listing 5

```
$ # invoke function via fn server
$ fn call mygo /oradev
{"message":"Hello World"}
```

Listing 6

```
$ # invoke function with UNIX curl
$ curl localhost:8080/r/mygo/oradev
{"message":"Hello World"}
```

Listing 7

```
$ # run the docker image
$ docker run oradev:0.0.2
{"message":"Hello World"}
```

Listing 8

```
$ docker run --rm -it --link fnserver:api -p 4000:4000 -e "FN_API_
URL=http://api:8080" fnproject/ui

> FunctionsUI@0.0.21 start /app
> node server

Using API url: api:8080
Server running on port 4000
```

Listing 9

Listing 1 zeigt das Erstellen eines neuen Verzeichnisses, in dem mit der Go-Sprache eine lokale Fn-Funktion initialisiert wird. Anschließend lässt sich die Funktion sofort mit dem Run-Kommando ausführen und die Ausgabe der HelloWorld-Anwendung betrachten *(siehe Listing 2)*.

Die erzeugte Datei zeigt, warum das möglich war. Es gibt eine Konfigurationsdatei namens „func.yaml“, die die Versionsnummer und die Laufzeit von Go spezifiziert. Außerdem wird eine Standard-Go-Routine („func.go“) mit einer Testdatei „test.json“ erzeugt *(siehe Listing 3)*. Wenn man die Ausgabe beim Ausführen der Funktion genau betrachtet, ist zu sehen, dass ein Docker-Image „oradev:0.0.1“ erstellt wurde.

Das Kommando „fn run“ ruft diese Funktion direkt auf. Um einen Endpunkt für die Funktion zu erstellen, ist als Erstes der Fn Server mit „$ fn start“ in einem anderen Terminal zu starten. Wenn der Server läuft, lässt sich die Funktion mit dem Kommando in *Listing 4* installieren. Der Name der Funktion wird aus dem Ordnernamen entnommen. Optional kann man den Namen auch in der Datei „func.yaml“ angeben.

Bei der Installation der Funktion erhöht sich die Version des Docker-Image auf 0.0.2. Durch das „Deploy“-Kommando wurde eine neue Anwendung registriert. Darüber hinaus ist ein Endpunkt für die Funktion entstanden. Um die Erstellung des Endpunkts zu prüfen, führt man die zwei Fn-Kommandos zum Auflisten der Anwendungen und der neuen Route aus *(siehe Listing 5)*.

Die Funktion ist jetzt am Fn Server registriert, der wie ein Mikro-API-Gateway agiert. Er nimmt Anfragen an die angeführten Endpunkte entgegen und ruft die installierten Funktionen auf. *Listing 6* zeigt, wie man das selbst ausprobieren kann. Wahlweise lässt sich die Funktion auch mit einem einfachen „curl“-Kommando von der Unix-Kommandozeile aufrufen, da Fn Server eine URL für die Funktion bereitstellt *(siehe Listing 7)*. Eine weitere Möglichkeit ist das direkte Ausführen des Docker-Image *(siehe Listing 8)*. Alle drei Ansätze liefern identische Ergebnisse.

Container/Funktionsdualität

Es ist zu beachten, dass ein Ausführen der Funktion über den URL-Endpunkt und das Ausführen des Docker-Containers das glei-

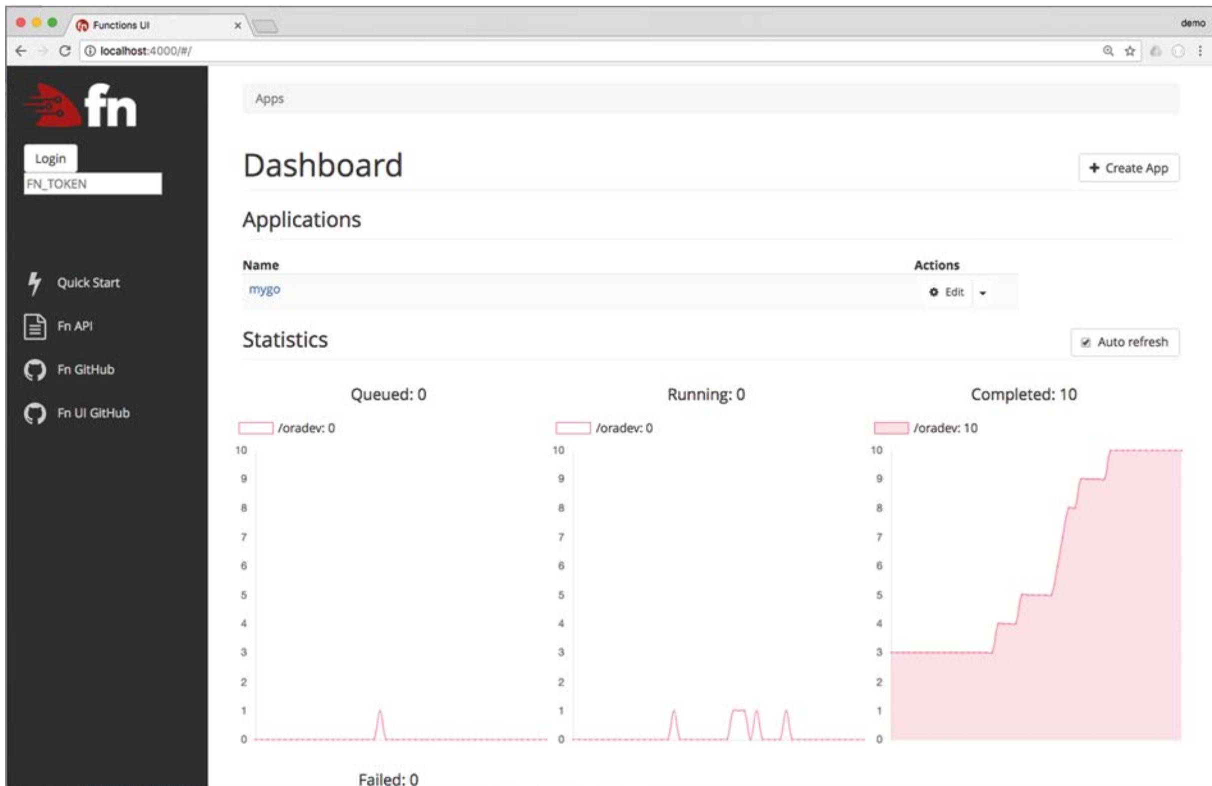

Abbildung 1: Die Überwachungskonsole

che Ergebnis liefern! Das Docker-Image, das die Funktion enthält, wurde automatisch erstellt, ohne irgendeine zusätzliche Konfiguration oder ein Kommando. Fn Project gibt so gesehen das Docker-Image kostenlos dazu. Zwei der vielen Vorzüge bei der Verwendung von Docker sind:

- Docker-Images lassen sich in jeder Public Cloud verwenden.
- Docker Hub kann Docker-Images speichern. Funktionen können somit im Docker Hub gespeichert werden.

Diese beiden Konzepte werden nachfolgend genauer betrachtet. Mit Fn lässt sich also einfach eine Funktion schreiben, ohne dabei auf Docker zu achten. Dennoch besteht der Vorteil, dass die Funktion in einem Container läuft.

```
$ curl localhost:8080/metrics | head

# HELP fn_api_completed Completed requests by path
# TYPE fn_api_completed counter
fn_api_completed{app="mygo",path="/oradev"} 11
# HELP fn_api_queued Queued requests by path
# TYPE fn_api_queued gauge
fn_api_queued{app="mygo",path="/oradev"} 0
# HELP fn_api_running Running requests by path
# TYPE fn_api_running gauge
fn_api_running{app="mygo",path="/oradev"} 0
# HELP fn_docker_stats_cpu_kernel docker_stats metric cpu_kernel
...
```

Listing 10

```
$ cd ~ && mkdir javatest && cd javatest
$ fn init --runtime java
Runtime: java
Function boilerplate generated.
func.yaml created.
```

Listing 11

Fn Monitoring

Fn Project bietet außerdem ein einfaches Werkzeug zur Überwachung, das als Docker-Container gestartet werden kann *(siehe Listing 9)*. Um auf die Konsole zuzugreifen, öffnet man einen Browser und verbindet sich mit Port 4000. Um eine Veränderung in den Kurven der Überwachungskonsole zu sehen, startet man die Go-Funktion einige Male *[9] (siehe Abbildung 1)*.

Prometheus Monitoring

Für eine etwas anspruchsvollere Überwachungslösung ist Prometheus von der Cloud Native Computing Foundation (CNCF) zusammen mit CNCF Grafana eine gute Wahl. Fn exportiert Metriken, die eine Überwachung mit Prometheus ohne zusätzliche Konfiguration zulassen. Auch ohne die Installation von Prometheus lassen sich die Metriken, die für Prometheus exportiert werden, mit

der URL „/metrics“ betrachten *(siehe Listing 10)*.

Weitere Einzelheiten zu Prometheus und Fn sind unter *[10]* beschrieben. Man kann auch mit dem Go-Beispiel ein HelloWorld-Beispiel in Java erstellen, indem der Laufzeitschalter auf „Java“ gestellt wird *(siehe Listing 11)*. Java 9 ist die Standard-Java-Version. Es ist zu beachten, dass für ein Java-Projekt auch eine Maven-Datei „pom.xml“ und ein Unit-Test „HelloFunctionTest.java“ erstellt werden *(siehe Listing 12)*.

```
$ tree

.
├── func.yaml
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── com
    │           └── example
    │               └── fn
    │                   └── HelloFunction.java
    └── test
        └── java
            └── com
                └── example
                    └── fn
                        └── HelloFunctionTest.java
```

Listing 12

JSON-Parameter-Bereitstellung und Funktionslogik

Um einige erweiterte Eigenschaften von Fn zu zeigen, verlassen wir das HelloWorld-Beispiel in Java und betrachten ein Beispiel für eine Recommendation-Engine. Man kann das Beispiel mit dem Kommando in *Listing 13* aus GitHub holen. Das API der Funktion simuliert die Empfehlungslogik. Sie verwendet ein POJO als Eingabe-Parameter. Dieser definiert das Alter des Reisenden, das Ziel und den Monat der Reise *(siehe Listing 14)*.

```
$ git clone https://github.com/fmunz/fn-recommend.git
$ cd fn-recommend
```

Listing 13

```
# check the API of the handler function
$ grep handle src/main/java/com/munzandmore/fn/RecommendFunction.java
  public String handleRequest(Traveller t) {
# examine the Traveller POJO
$ cat src/main/java/com/munzandmore/fn/Traveller.java
package com.munzandmore.fn;
public class Traveller {
  public Integer age ;
  public String destination ;
  public String month;
}
```

Listing 14

Dieses Beispiel wird automatisch im Docker Hub eingecheckt – im Gegensatz zum vorherigen Go-Beispiel, das nur lokal vorgehalten wurde. Zu diesem Zweck wird die Umgebungsvariable „FN_REGISTRY“ auf die „DOCKER_ID“ gesetzt und außerdem im Docker Hub eingeloggt. Man ersetzt also im Beispiel „DOCKER_ID“ durch sein eigenes Docker-Log-in *(siehe Listing 15)* und installiert anschließend die Funktion. Die Funktion wird für eine Abenteuerreise-Anwendung verwendet, daher auch der Name *(siehe Listing 16)*.

```
# set environment for Docker hub
$ export FN_REGISTRY=YOUR_DOCKER_ID
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If
you don't have a Docker ID, head over to https://hub.docker.com to cre-
ate one.
Username: DOCKER_ID
Password:
Login Succeeded
```

Listing 15

Aus der Ausgabe ist zu erkennen, dass das Docker-Image erstellt und im Docker Hub unter „DOCKER_ID/fn-recommend:0.0.2“ eingecheckt wurde. Man kann wie gewohnt die neue Anwendung und die neue Route im Fn Server prüfen *(siehe Listing 17)* und die Funktion mit einem „POST“-Request über ein „Curl“-Kommando ausführen, wenn man die erforderliche JSON-Datenstruktur für den Request liefert. Standardmäßig verwendet Fn das Jackson-Java-Framework, um automatisch JSON-Eingabeparameter für den korrekten Java-Typ bereitzustellen. Man kann aber auch jedes beliebige

```
$ fn deploy --app advtravel
Deploying fn-recommend to app: advtravel at path: /fn-recommend
Bumped to version 0.0.2
Building image DOCKER_ID/fn-recommend:0.0.2
Pushing DOCKER_ID/fn-recommend:0.0.2 to docker registry...The push re-
fers to repository [docker.io/DOCKER_ID/fn-recommend]
7e2c18073a13: Layer already exists
…
0.0.2: digest: sha256:549e492a08d924dcfeef5f0354dc7d2df57cba820bcfa7ec5
50a1779a173983c size: 1997
Updating route /fn-recommend using image DOCKER_ID/fn-recom-
mend:0.0.2...umped to version 0.0.2
```

Listing 16

Framework zur Bereitstellung von JSON oder anderen Formaten wie XML etc. verwenden *(siehe Listing 18)*.

Für die Evaluierung unterschiedlicher Eingabe-Parameter ist ein grafisches Tool wie Postman geeigneter. *Abbildung 2* zeigt, was die Fn-basierte Beispiel-Anwendung für eine Reise nach Sydney empfiehlt. Die Ausgabe sollte wie in *Abbildung 3* aussehen.

Fn in Public Clouds (IaaS)

Eine häufige Frage ist, wie Fn Project als Cloud-agnostisches Framework in Public Clouds verwendet werden kann. Ähnlich wie bei der lokalen Installation, die in den Beispielen gezeigt wurde, kann es auf jeder Public Cloud IaaS installiert werden. Bei den meisten IaaS-Clouds reicht es aus, das Installations-Kommando direkt bei der Erstellung einer Rechner-Instanz als sogenannte „user data" zu übergeben, also als Kommandos, die ausgeführt werden, wenn die Instanz bereitgestellt wird. Beim Betrieb in einer Public Cloud ist außerdem darauf zu achten, Zugriffsregeln für den Fn Server auf Port 8080 einzurichten, entweder von der eigenen IP oder von allen öffentlichen IP-Adressen ausgehend.

Wer Fn Project auf einem IaaS betreibt, bekommt allerdings nicht den Vorteil des echten „Bezahlens pro Aufruf". Dies wäre nur mit einem echten FaaS-Service eines Cloud-Providers der Fall. Die Funktionen selbst laufen dennoch aus Anwendersicht serverlos in einer standardisierten, portablen und skalierbaren Art und Weise. Wenn der Fn Server beim Cloud-Provider seiner Wahl läuft, gibt es zwei Möglichkeiten, die Recommendation-Engine aus dem Beispiel zu installieren *(siehe Listing 19)*.

Es ist zu beachten, dass mit den beiden Kommandos nie die Funktion oder das Container-Image auf die Cloud-Instanz zu kopieren war. Wenn die Funktion das erste Mal aufgerufen wird, checkt Fn den Docker-Container, speichert ihn lokal

```
$ fn apps list
advtravel
mygo


$ fn routes list advtravel
path       image          endpoint
/fn-recommend DOCKER_ID/fn-recommend:0.0.2 localhost:8080/r/advtravel/
fn-recommend
```

Listing 17

```
$ cat testdata/muc.json
{
   "age": 41,
   "destination": "Munich",
   "month": "Oct"
}
# get a recommendation for Munich in October
$ curl -X POST --data @testdata/syd.json localhost:8080/r/advtravel/fn-
recommend

Visit the Octoberfest!
# there is more test data under testdata/Casablanca.json
# see what is recommended for that city!
```

Listing 18

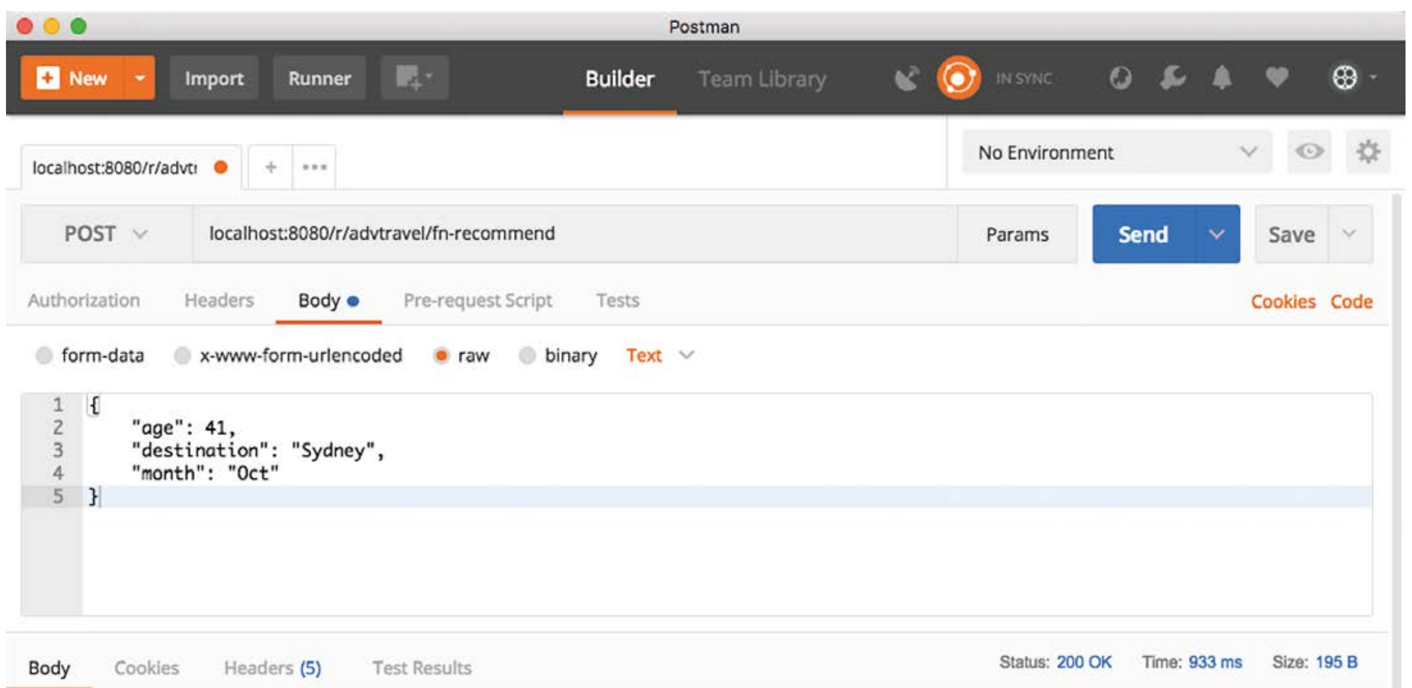

Abbildung 2: Eine Fn-basierte Beispiel-Anwendung

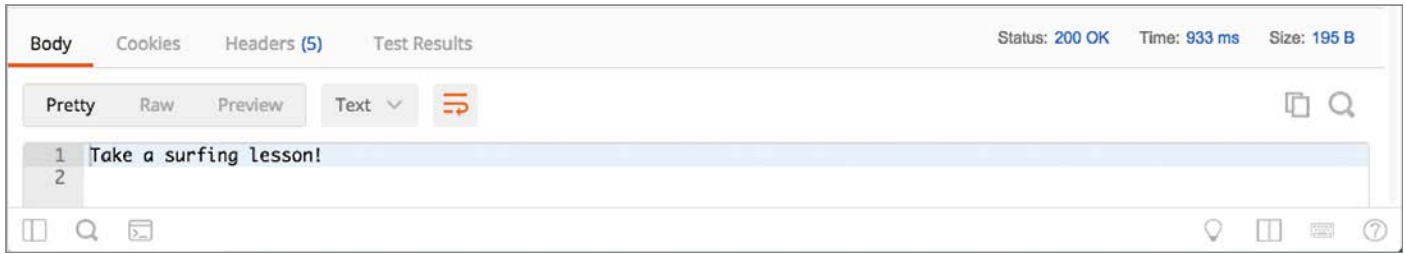

Abbildung 3: Die Ausgabe mit Postman

```
# example 1 (for demo purpose only, in production use approach below)
# note: run these commands on the cloud instance
$ fn apps create advtravel
$ fn routes create advtravel /fn-recommend DOCKER_ID/fn-recommend:0.0.2

# check for the created route
$ fn routes list advtravel
```

Listing 19

```
# example 2
# run these commands on cloud instance


$ export FN_API_URL=URLCloudInstance
$ fn deploy --app advtravel
$ fn routes list advtravel
```

Listing 20

```
$ curl -X POST --data @testdata/syd.json PUBLIC_IP:8080/r/advtravel/
fn-recommend
```

Listing 21

und führt anschließend die Funktion aus. Eine bessere Methode zur Installation der Funktion ist, die „FN_API_URL"-Umgebungsvariable lokal zu konfigurieren, sie auf die Remote-Cloud-Instanz zu setzen und anschließend das lokale Fn-Installationskommando gegen die Remote-Cloud-Instanz auszuführen *(siehe Listing 20)*.

Wenn Fn in der Cloud läuft und die Anwendung installiert ist, kann man von jedem lokalen Rechner aus über die Kommandozeile oder mittels Postman auf die Anwendung zugreifen. Der Aufruf erfolgt genauso wie im lokalen Beispiel, man ersetzt einfach „localhost" durch die öffentliche IP-Adresse der Cloud-Instanz *(siehe Listing 21)*. Eine aufgezeichnete Demo von der Devoxx-Konferenz zur Installation einer Fn-basierten Anwendung mit der Recommendation-Engine auf IaaS steht unter *[11]*.

JAX-RS, Spring Cloud und mehr

Da Fn Project nur Docker als Abhängigkeit hat und die Maven-Datei „pom.xml" auch generiert wird, kann die Funktionsentwicklung einfach um weitere Java-Frameworks erweitert werden. Bisher wurde im Fn-Team an der Unterstützung von JAX-RS mit Fn Projects gearbeitet *[12]*. Spring unterstützt ebenfalls die Implementierung von Anwendungslogik als Funktionen mit ihrem Konvention-vor-Konfiguration-Ansatz. Spring-Cloud-Funktionen lassen sich mit Fn verwenden *[13, 14]*.

Fn LB, eine separate Komponente, ist für Lastverteilung und intelligentes Routing zuständig. Werden Funktionen als „hot functions" installiert, bleibt ein Container für 30 Sekunden bestehen und wird nicht für jeden Aufruf neu gestartet. Fn LB leitet dann Aufrufe dorthin, um optimale Performance zu sichern *[15]*.

Am Anfang dieses Artikels haben wir uns mit den Unterschieden zwischen Microservices und FaaS befasst und erklärten, dass ein Microservice typischerweise mehr als eine einzige Methode oder Funktion enthält. Heutzutage werden häufig grafische Tools oder übergeordnete PaaS-Dienste verwendet, um FaaS in sinnvolle, größere Services zu integrieren. Diese grafischen Tools bieten jedoch meistens keinen wirklich guten Einblick in die Einzelheiten des übergeordneten Service. Erkenntnisse aus Projekten mit ESB und BPEL zeigen, dass diese Details nicht gleichzeitig grafisch angezeigt werden können und deswegen häufig unter einem Eigenschaften-Tab des grafischen Modells versteckt sind. Einen solchen „Flow" in einem grafischen Modell darzustellen, ist daher häufig eingeschränkt.

Fn Flow behandelt dieses Problem für Fn Project. Es verfolgt einen interessanten „Code first"-Ansatz, indem das Java-8-CompletableFutures-API mit Methoden wie „thenApply()" oder „thenCompose()" etc. verwendet wird. Grafische Tools oder große YAML-Dateien sind nicht erforderlich; die Verkettung der Funktionen erfolgt ausschließlich mit Java-8-Konstrukten und ist dadurch leicht lesbar. Eine interessante Anwendung dieses Konzepts wird bei der Verwendung von SAGAs anstelle einer ACID-Transaktion für eine Reisebuchungsanwendung *[16]* basierend auf Microservices gezeigt *(siehe Abbildung 4)*.

Was auf den ersten Blick wie ein gewöhnliches Java-8-Programm aussieht, erinnert bei der Ausführung eher an Apache Spark. Die Ausführung findet parallel statt, die Eingabe-Parameter der Funktion werden übergeben und Rückgabe-Werte entsprechend zurückgegeben. Jede Funktion wird in seinem eigenen Container unter Verwendung von Verbindung, Fehler-

```
public void book1(TripReq input) {
    Flow f = Flows.currentFlow();

    FlowFuture<BookingRes> flightFuture =
        f.invokeFunction("./flight/book", input.flight, BookingRes.class);

    FlowFuture<BookingRes> hotelFuture =
        f.invokeFunction("./hotel/book", input.hotel, BookingRes.class);

    FlowFuture<BookingRes> carFuture =
        f.invokeFunction("./car/book", input.carRental, BookingRes.class);

    flightFuture.thenCompose(
        (flightRes) -> hotelFuture.thenCompose(
            (hotelRes) -> carFuture.whenComplete(
                (carRes, e) -> EmailReq.sendSuccessMail(flightRes, hotelRes, carRes)
            )
        )
    );
}
```

Abbildung 4: Der Einsatz von SAGAs anstelle einer ACID-Transaktion

behandlung und Fan In/Out ausgeführt. Fn Flow kann die Aufrufkurven nachverfolgen und sie visualisieren.

Fn on Kubernetes

Auf der KubeCon wurde im Dezember 2017 offiziell Support für Fn on Kubernetes angekündigt. Fn kann mit Helm, einem Paketmanager für Kubernetes, installiert werden. Vorkonfigurierte Pakete wie Fn Service, Fn UI, Flow Service und Flow UI werden als Helm-Chart bereitgestellt. Mit diesem lässt sich Fn auf jedem beliebigen Kubernetes-Cluster installieren *[18]*. Es ermöglicht außerdem den Betrieb von Fn auf dem brandneuen Oracle-managed Kubernetes Service, Oracle Container Engine (OCE) oder einer lokalen Installation von Minikube auf dem Laptop *[19, 20]*.

Fazit

Fn Project ist ein interessanter neuer Ansatz für die serverlose Welt. Es ist Cloud-agnostisch und verhindert dadurch den Anbieter-Lock-in. Darüber hinaus sind Entwickler bei der Verwendung von Fn nicht an bestimmte Programmiersprachen gebunden. Funktionen werden automatisch als Docker-Image gebaut, ohne dass der Entwickler dafür zusätzlich etwas tun muss. Die Funktionen können überall laufen und werden vom Docker Hub geladen.

Fn ist in die Projekte der Cloud Native Computing Foundation eingebunden und bietet Support für Kubernetes und Prometheus; hoffentlich kommen noch weitere dazu. Zu guter Letzt wird es interessant sein zu sehen, ob Oracle oder ein anderer Cloud-Provider einen Fn-basierten FaaS als PaaS in naher Zukunft mit Bezahlung pro Aufruf und voll automatisierter Skalierung anbieten werden *[21]*.

Referenzen

[1] Oracle Developer: *https://developer.oracle.com/en_US/opensource/serverless-with-fn-project*
[2] Referenz Expedia: *www.youtube.com/watch?v=gT9x9LnU_rE*, Referenz Postlight: *https://goo.gl/uAMGYF*
[3] Funktionen vs. Container: *https://medium.com/oracledevs/containers-vs-functions-51c879216b97*
[4] Microservices, M. Fowler: *https://martin-fowler.com/articles/microservices.html*
[5] Mark Cavage, Serverless, Java One Keynote 2017: *https://www.youtube.com/watch?v=UNg9lmk60sg*
[6] Overview FaaS Frameworks: *https://github.com/faas-lane/ FaaS-Lane/tree/master/candidates*
[7] Serverless Classification: *http://www.munzandmore.com/2018/cc/serverless-faas-frameworks-and-platform-classification*
[8] Fn Project Installation: *https://github.com/fnproject/fn*
[9] Fn UI: *https://github.com/fnproject/ui*
[10] Fn Monitoring mit Prometheus: *https://medium.com/fnproject/announcing-prometheus-metrics-from-fn-2d0f9ddf0f09*
[11] Fn Project on Public Clouds (IaaS): *http://www.munzandmore.com/2017/aws/fn-project-on-public-clouds*
[12] Fn mit JAX-RS: *https://github.com/fnproject/fn-jrestless-example*
[13] Spring-Cloud-Funktionen: *https://spring.io/blog/2017/07/05/introducing-spring-cloud-function*
[14] Fn with Spring Cloud Functions: *https://medium.com/fnproject/announcing-spring-cloud-function-support-for-fn-project-921e54f49d99*
[15] Fn LB: *https://github.com/fnproject/fn/tree/master/fnlb*
[16] Serverless Sagas mit Fn Flow: *https://medium.com/fnproject/serverless-sagas-with-fn-flow-d8199b608b12*
[17] Acht Gründe, warum wir Fn Project entwickelt haben: *https://medium.com/fnproject/8-reasons-why-we-built-the-fn-project-bcfe45c5ae63*
[18] Fn Project mit Helm Chart für Kubernetes: *https://medium.com/fnproject/fn-project-helm-chart-for-kubernetes-e97ded6f4f0c*
[19] Fn Project on Kubernetes: *https://blogs.oracle.com/developers/kubernetes-server-less-and-federation-oracle-at-kubecon-2017*
[20] Oracle Container Engine: *https://blogs.oracle.com/developers/announcing-oracle-container-engine-and-oracle-container-registry-service*
[21] Serverlos und Fn Präsentationen: *http://www.munzandmore.com/2017/cc/devoxx-2017-serverless*

Dr. Frank Munz
fm@munzandmore.com