Serverless Java mit Fn Project

Marcel Amende, Oracle Deutschland B.V. & Co. KG

Das Serverless-Computing-Programmiermodell ist der neue Trend beim Entwickeln in der und für die Cloud. Man braucht sich dabei keine Gedanken mehr über Server, Storage, Netzwerk, virtuelle Maschine und selbst zu implementierende, infrastrukturelle Hilfsroutinen zu machen und kann sich ganz auf die Hauptsache konzentrieren: den eigenen Code, der die Geschäfts-Anforderungen modular in Form von kleinen Funktionspaketen (Function-as-a-Service, FaaS) umsetzt. Server gibt es beim Serverless Computing natürlich dennoch. Sie bleiben allerdings für den Entwickler unsichtbar und brauchen selbst bei sich ändernden Last-Anforderungen keinerlei Administration.

Mit AWS Lambda *[1]*, Cloud Functions *[2]* und Azure Functions *[3]* gibt es bereits einige kommerzielle, weitgehend geschlossene Serverless-Computing-Angebote am Markt. Fn Project ist eine Open-Source-Serverless-Plattform als Alternative, die zudem erstklassige Unterstützung für die Programmiersprache Java bietet. Auch Fn Project hat bereits eine längere Historie. Das Team dahinter gehört zu den Pionieren des Serverless Computing und rekrutiert sich aus den Entwicklern der IronFunctions-Serverless-Plattform *[4]*.

Im Unterschied zu den erstgenannten Plattformen bekommt man bei Fn Project nicht nur ein API oder ein Benutzer-Interface, um den Code hochzuladen; Fn Project ist komplett Container-basiert. Die Funktion ist hier der Container und der Container ist die Funktion. Dadurch ist sie unverändert überall lauffähig: als Docker-Container auf dem Laptop des Entwicklers, auf einer Container-Plattform im eigenen Rechenzentrum oder als Kubernetes-Cluster in der Cloud jedes beliebigen Anbieters.

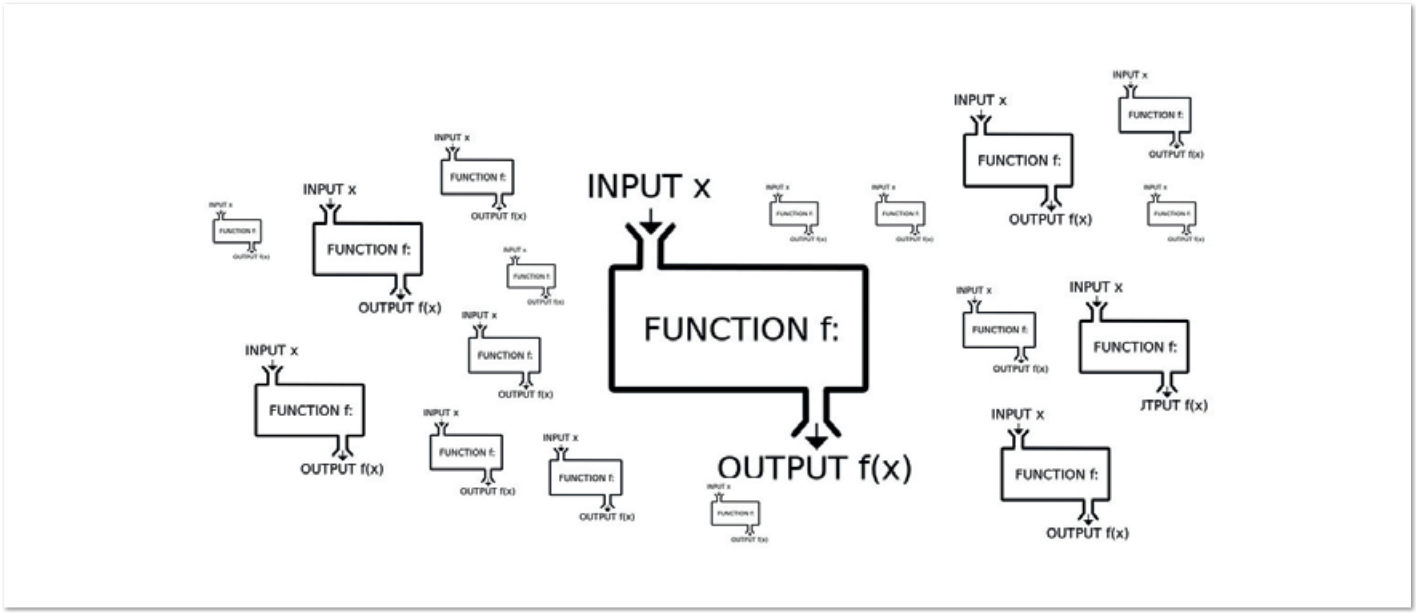

Abbildung 1: Funktionen im Serverless-Computing-Programmiermodell

Einführung in Fn Project

Der Fokus von Fn Project liegt auf der einfachen Nutzbarkeit. Es ist in Go implementiert, kann aber mit jeder beliebigen Programmiersprache genutzt und einfach um zusätzliche erweitert werden.

Funktionen sind das Herzstück der Serverless-Computing-Plattform. Sie stellen kleine, isolierte Codefragmente dar, die eine bestimmte Anforderung möglichst optimal erledigen. Sie lesen von der (oft Standard-)Eingabe, tätigen ihre Berechnungen und erzeugen eine Ausgabe *(siehe Abbildung 1)*.

Die Funktion stellt auch die Ausbringungseinheit dar. Um alles weitere, die Provisionierung der benötigten Infrastruktur, die Ausbringung, Skalierung, Verfügbarkeit, Elastizität, Sicherheit und Abrechnung, kümmert sich die Plattform. Code lässt sich in dieser Form sehr stark verdichtet betreiben, was letztendlich Kosten spart.

Bei Fn Project entwickelt man Funktionen von der Kommandozeile aus durch Nutzung des Kommandos „fn". Einzige Abhängigkeit ist Docker. Dieses muss vorinstalliert sein, da jede Funktion in Form eines Docker-Containers erstellt und ausgebracht wird. Für die einfache Erstellung von Funktionen gibt es bereits eine Reihe von Entwicklungsbaukästen (FDK, Function Development Kit) für Java, Go, JavaScript, Ruby, Rust oder Python.

Den größten Funktionsumfang und die größte Robustheit hat derzeit das Java FDK, das in enger Zusammenarbeit mit dem Java-Entwicklerteam von Oracle entsteht. Ein FDK ist ein vorgefertigter Do-

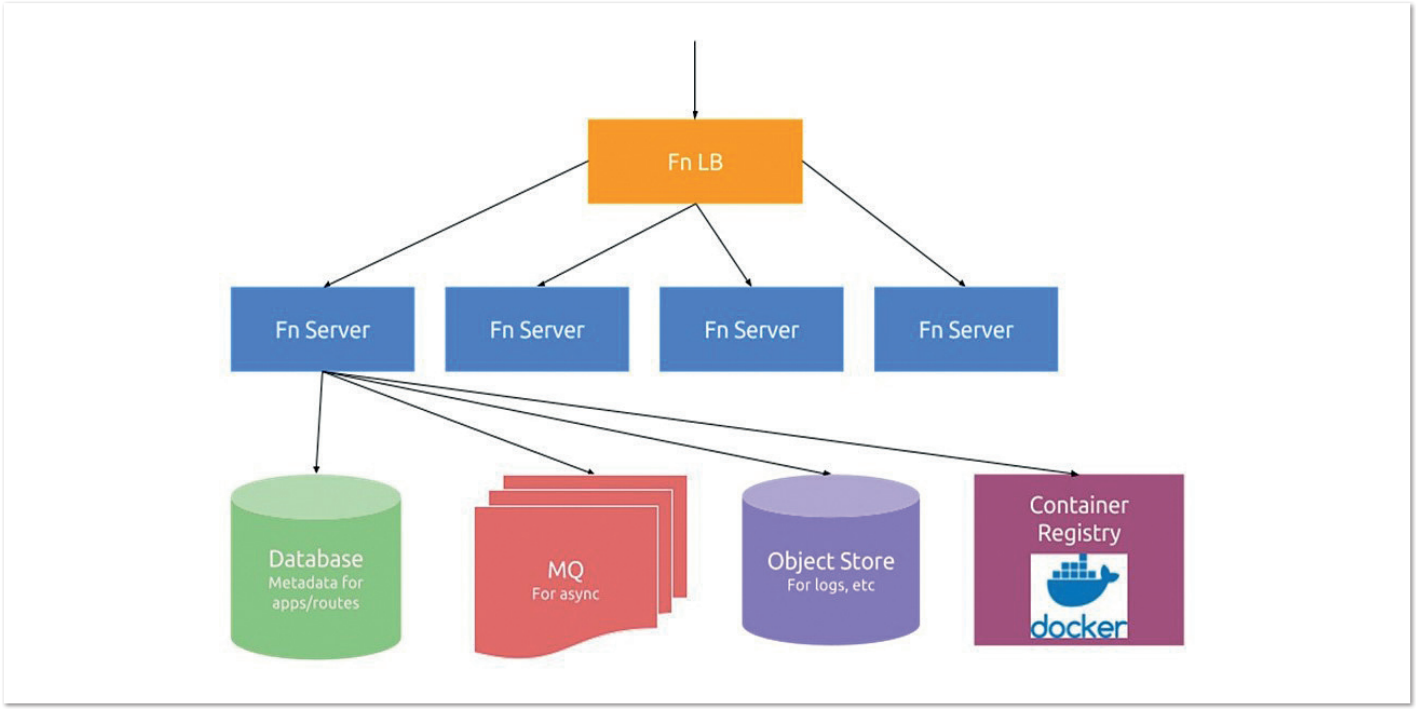

Abbildung 2: Architektur von Fn Project

cker-Container mit den benötigten Laufzeit-Komponenten, in den der Funktionscode eingebracht wird. Aber auch jeder selbst erstellte Docker-Container kann als Funktion betrieben werden. Dadurch ist es möglich, Lambda-Funktionen lokal auszuführen oder auf Cloud-Plattformen anderer Anbieter zu migrieren.

```
$ fn start
        ______
       / ____/___
      / /_  / __ \
     / __/ / / / /
    /_/   /_/ /_/
        v0.3.313

time="2018-03-03" msg="Fn serving on `:8080`" type=full
```

Listing 1

```
$ fn init --runtime=java javaaktuell
Creating function at: /javaaktuell
        ______
       / ____/___
      / /_  / __ \
     / __/ / / / /
    /_/   /_/ /_/

Runtime: java
Function boilerplate generated.
func.yaml created.
$
```

Listing 2

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

Listing 3

```
$ fn run
Building image javaaktuell:0.0.1
Sending build context to Docker daemon  13.82kB
Step 1/11 : FROM fnproject/fn-java-fdk-build:jdk9-
latest as build-stage
...
Step 11/11 : CMD ["com.example.fn.HelloFunction::handle
Request"]
 ---> Using cache
 ---> 79717de5757b
Successfully built 79717de5757b
Successfully tagged javaaktuell:0.0.1
Hello, world!
```

Listing 4

Zusätzlich gibt es noch eine Reihe weiterer architektonischer Grundkomponenten, die für den Betrieb von Fn Project notwendig sind *(siehe Abbildung 2)*. Zentral ist der Fn Server für die Ausführung der Funktionen und eine lokale oder zentrale Container Registry für deren Ausbringung. Für die verteilte Nutzung und Protokoll-Adaption nutzt man vor den Server-Komponenten einen Load Balancer. Bei asynchronen Funktionen wird eine Message Queue (etwa auf Basis von Redis oder Bolt) benötigt. Hinzu kommt eine Datenbank (wie MySQL, Progress oder sqlite3) für persistente Operationen, die aus Latenzgründen aber möglichst wenig und schreibend nur außerhalb der Anfragezyklen einer Funktion genutzt wird.

Erste Schritte mit Fn Project

Als Open-Source-Projekt unter Apache 2.0 Lizenz findet man den Fn-Project-Quellcode und die direkt ausführbare Variante auf GitHub *[5]*. Auch die Schnellstart-Anleitung und Anwendungsbeispiele sind hier veröffentlicht. Weitere, ausführliche Tutorien finden sich auf der Projektseite „fnproject.io“ *[7]*. Auf MacOS lässt sich Fn Project mit dem Paketmanager „Homebrew“ per „$ brew install fn“ installieren. Alternativ können die Installationsroutinen für die Kommandozeilen-Ausführung unter MacOS, Linux und Windows von GitHub bezogen werden *[6]*. Nach der Installation lässt sich der Fn Server starten, um später Funktionen lokal ausführen und testen zu können *(siehe Listing 1)*.

Eine erste Java-Funktion

Die Erstellung einer einfachen Funktion aus einer Vorlage dauert mit dem Kommando „fn init“ nur wenige Sekunden. Die gewünschte Laufzeit-Umgebung übergibt man als Parameter. In dem Beispielprojekt dieses Artikels wird eine auf Java 9 basierende Funktion entwickelt, daher heißt der Parameter „Java“. Mit derzeit unter anderem Go, Node, Python, Ruby oder Rust ist gleichsam eine Reihe weiterer Laufzeit-Umgebungen unterstützt *(siehe Listing 2)*. Die Initialisierungsroutine erzeugt eine Projektverzeichnisstruktur mit vier Dateien *(siehe Listing 3)*.

Eine Datei „func.yaml“ wird unabhängig von der gewählten Laufzeit-Umgebung immer erzeugt. Im Falle einer Java-Funktion enthält diese Parameter für Laufzeit und Version sowie die vollqualifizierten Namen der Java-Funktionsklasse und der aufzurufenden Methode. Zusätzlich findet man den Quellcode einer Beispielfunktion „HelloFunction.java“ mit einem JUnit Testcase „HelloFunctionTest.java“ und eine Maven-„pom.xml“-Datei zum Kompilieren und Testen der Funktion.

Build und Run

Im nächsten Schritt wird der Build-Prozess für die Funktion gestartet, um diese dann lokal zu testen. Dafür kommt zunächst die lokale Docker-Registry zum Einsatz, deren Benutzername mit „$ export FN_REGISTRY=<local_docker_user>“ als Umgebungsvariable gesetzt wird.

Der erste Build- und Deployment-Prozess dauert länger als nachfolgende, da die benötigten Docker-Basis-Images aus dem Docker-Hub geladen werden. Gleiches gilt für die Verwendung von Maven, bei dem zusätzliche Java-Packages geladen werden. Mit dem Befehl „fn run“ lassen sich der Build-Prozess der Funktion starten und diese sofort einmalig ausführen *(siehe Listing 4)*.

```
$ echo "Java aktuell" | fn run
Building image javaaktuell:0.0.1
...
Successfully built 79717de5757b
Successfully tagged javaaktuell:0.0.1
Hello, Java aktuell!
```

Listing 5

```
package com.example.fn;

public class HelloFunction {

    public String handleRequest(String input) {
        String name = (input == null || input.isEmpty())
                ? "world"  : input;
        return "Hello, " + name + "!";
    }
}
```

Listing 6

```
    // My POJO
    public static class MyCustomer(){
        String name;
    };
...
    // Function Method
    public String handleRequest(MyCustomer cust){
        ....
    }
```

Listing 7

```
$ fn build
Building image javaaktuell:0.0.1
...
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.example.fn.HelloFunctionTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.941 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
...
Function javaaktuell:0.0.1 built successfully.
$
```

Listing 8

Im Hintergrund wurde die Funktion mit Maven kompiliert, in einen Docker-Container eingepackt, an die lokale Docker-Registry übergeben und ausgeführt. Die Funktion gibt den Text „Hello, world!" auf der Standard-Ausgabe aus. Man kann der Aufruf-Funktion per Pipe-Befehl auch Parameter über die Standardeingabe übergeben *(siehe Listing 5)*.

Die Funktion hat offensichtlich den Text „Java aktuell" von der Standard-Eingabe gelesen und daraufhin die Ausgabe „Hallo, Java aktuell!" generiert. Ein Blick in den Quellcode der generierten Klasse „com.example.fn.HelloFunction" zeigt, dass es sich bei der Funktion um eine Methode „handleRequest" in einem einfachen Java-POJO handelt *(siehe Listing 6)*.

Die Methode akzeptiert einen String und gibt ebenfalls einen String zurück. Das Java-FDK unterstützt allerdings auch das Binden anderer Ein- und Ausgabe-Typen wie Streams, primitive Daten-Typen, Byte-Arrays und in POJOs gewandelte JSON-Objekte. Will man JSON verwenden, ersetzt man die Aufruf- beziehungsweise Rückgabe-Argumente der Funktionsmethode einfach durch POJOs und setzt in der Datei „func.yaml" den Parameter „Format" zu „format: json" *(siehe Listing 7)*.

Das Java-FDK bindet basierend auf diesen Argumenten die Eingabedaten automatisch. Diese JSON-Unterstützung ist bereits in das FDK eingebaut. Wandler für andere Datenformate, wie XML oder Protobuf, lassen sich einfach ergänzen.

Zum generierten Beispiel gehört mit der Klasse „com.example.fn.HelloFunctionTest" auch ein JUnit-Test. Dieser wird von Maven ausgeführt, wenn man den Build per „fn build" explizit startet *(siehe Listing 8)*.

Mit Fn Server Funktionen ausbringen

Nachdem der Funktions-Container erstellt ist, kann dieser über das Kommando „fn deploy" auf einem Fn Server ausgebracht werden. Generell können dafür lokale und entfernte beziehungsweise öffentliche Docker-Registraturen verwendet werden. Wir nutzen die lokale Registratur in Kombination mit dem bereits gestarteten, lokalen Fn Server. Das spart dem Entwickler viel Zeit und unterscheidet Fn Project signifikant von anderen Serverless-Computing-Frame-

```
$ fn deploy --local -app myapp
Deploying javaaktuell to app: javaapp at path: /javaaktuell
Bumped to version 0.0.2
Building image javaaktuell:0.0.2
...
Successfully built 031e7a11ede4
Successfully tagged javaaktuell:0.0.2
Updating route /javaaktuell using image javaaktuell:0.0.2...
$
```

Listing 9

```
$ curl --data "Test" http://localhost:8080/r/myapp/javaaktuell
Hello, Test!
$
```

Listing 10

works und Implementierungen. Diese sind meist nur als Services in der Cloud eines bestimmten Anbieters verfügbar und können lokal höchstens die Ausführung einer Funktion simulieren. Bei Fn Project bestehen keine Unterschiede zwischen lokalem Server und Service in der Cloud; man nutzt immer die exakt selbe Codebasis und Implementierung. Für das lokale Ausbringen wird die Option „--local" gesetzt *(siehe Listing 9)*.

Der Parameter „--app" dient dazu, mehrere Funktionen zu einem Dienst zusammenzuführen. In der letzten Zeile der Ausgabe sieht man, dass mit dem Ausbringen einer Funktion auf dem Fn Server automatisch eine Route mit der Funktion assoziiert wird, in diesem Fall mit „/javaaktuell" der Name des Funktions-Verzeichnisses. Dies bedeutet, dass die Funktion auch als Webservice über das Hilfskommando „curl" oder aus einem Browser aufrufbar ist *(siehe Listing 10)*.

Es fällt auf, dass die Ausführung dieser einfachen Funktion mit mehreren Hundert Millisekunden recht lange dauert. Dies liegt daran, dass die vorliegende Funktion eine sogenannte „Cold Function" ist, bei der die Funktion inklusive ihres Containers und ihrer Laufzeit-Umgebung für jeden Aufruf neu gestartet wird. Für einen Webservice, der oftmals ausgeführt wird, ist dies nicht praktikabel. Durch Setzen des Parameters „format: http" in der Datei „func.yaml" ändert man die Funktion in eine „Hot Function", die für das Bedienen mehrerer Anfragen bis zu einer voreingestellten Auszeit von dreißig Sekunden am Leben gehalten wird.

Asynchrone Funktionen

Asynchrone Funktionen eignen sich besonders für rechenintensive Aufgaben und Massen-Operationen. Die Funktions-Aufrufe werden in eine Warteschlange eingestellt und zu einem späteren Zeitpunkt ausgeführt. Der Aufrufer wartet nicht auf eine sofortige Rückantwort wie bei einer synchronen Funktion. Er erhält stattdessen ein JSON-Objekt mit einer Aufruf-ID. Das Ergebnis der eigentlichen Funktionsausführung wird in ein Log eingestellt. Die Aufruf-ID lässt sich später nutzen, um das Ergebnis aus dem Log abzurufen. Für die Erstellung einer asynchronen Funktion muss die „func.yaml" um das Attribut „type: async" ergänzt werden. Nach erneuter Ausbringung kann die asynchrone Funktion aufgerufen werden, in diesem Beispiel durch das Fn-Kommandozeilen-Interface *(siehe Listing 11)*.

Die Aufruf-ID wird in Form eines JSON-Objektes zurückgegeben. Ebenfalls über die Fn-Kommandozeile lässt sich per Status-Endpunkt-API der Status der Funktionsausführung abfragen *(siehe Listing 12)*.

Der Status „success" bedeutet, dass der Aufruf erfolgreich beendet wurde. Zusätzlich wird die Einstellungs-, Start- und Endzeit ausgegeben. Um zu überprüfen, ob die Funktion die richtige Ausgabe generiert hat, kann der Log-Endpunkt abgefragt werden, um das Ausgabeergebnis zu erhalten *(siehe Listing 13)*.

Komplexe Funktionen mit Fn Flow

Fn Flow stellt dem Entwickler ein flexibles Modell zur Verfügung, um einzelne Funktionen zu komplexeren Arbeitsabläufen zusammenzusetzen. Es behandelt Aspekte wie die Ausführungs-Reihenfolge, Parallelisierung, Wiederanläufe und Fehlerbehandlung. Es ist dabei kein weiteres schwergewichtiges, grafisches Workflow-Werkzeug, wie ein Service-Bus, ein BPEL- oder BPMN-Implementierungen sie

```
$ fn call javaapp /javaaktuell
{"call_id":"01C96TVVR247WHT00000000000"}
$
```

Listing 11

```
$ fn calls get javaapp 01C96TVVR247WHT00000000
ID: 01C96TVVR247WHT00000000000
App: javaapp
Route: /javaaktuell
Created At: 2018-03-22T12:27:47.330Z
Started At: 2018-03-22T12:27:48.829Z
Completed At: 2018-03-22T12:27:49.331Z
Status: success
$
```

Listing 12

```
$ fn logs get javaapp 01C96TVVR247WHT00000000000
HTTP/1.1 200 INVOKED
Content-Length: 13
Content-Type: text/plain

Hello, world!
$
```

Listing 13

```
Flow f = Flows.currentFlow();
FlowFuture flight = f.invokeFunction("/flight/book",
...);
FlowFuture hotel = f.invokeFunction("/hotel/book", ...);
flight.thenCompose(
    (flightRes) -> hotel.whenComplete(
        (hotelRes) -> Email.send(flightRes, hotelRes)
    ).exceptionallyCompose((e) -> cancel(/hotel/cancel))
).exceptionallyCompose((e) -> cancel(/flight/cancel));
```

Listing 14

darstellen, sondern wird im Code definiert und ganz natürlich als Funktion ausgeführt. Das zugehörige API ermöglicht Einsicht in den Status des Ablaufs, in Logs und Stack Traces. Auch wenn die erste Implementierung von Fn Flow für Java verfügbar ist, ist es konzeptionell sprachagnostisch.

Implementierungen in Go, JavaScript und Python werden in Kürze folgen. Ein typisches Beispiel ist die Buchung einer Reise, bei der folgender Ablauf denkbar wäre: Stelle parallel Buchungsanfragen für einen Flug und ein Hotel; verschicke im Erfolgsfall eine Bestätigungsmail und storniere das Hotel, wenn kein Flug verfügbar ist. Als Java-Code sieht dies in leicht vereinfachter Form wie in *Listing 14* aus.

Das hier verwendete asynchrone und verteilte Programmiermodell nutzt sogenannte „Futures" und „Promises". Ein Objekt wird mit einer bestimmten Ausführung starten, zu einem zukünftigen Zeitpunkt eine Ausgabe ermitteln oder einen Grund liefern, warum dies nicht erfolgen konnte. Registrierte „Callbacks" verarbeiten die Rückgaben. Die Klasse „Flow" übergibt jeden Schritt des Arbeitsablaufs an den Flow-Server, der die Schritte als individuelle Aufrufe an den FN-Server orchestriert und auch die Rückantworten auswertet. Den

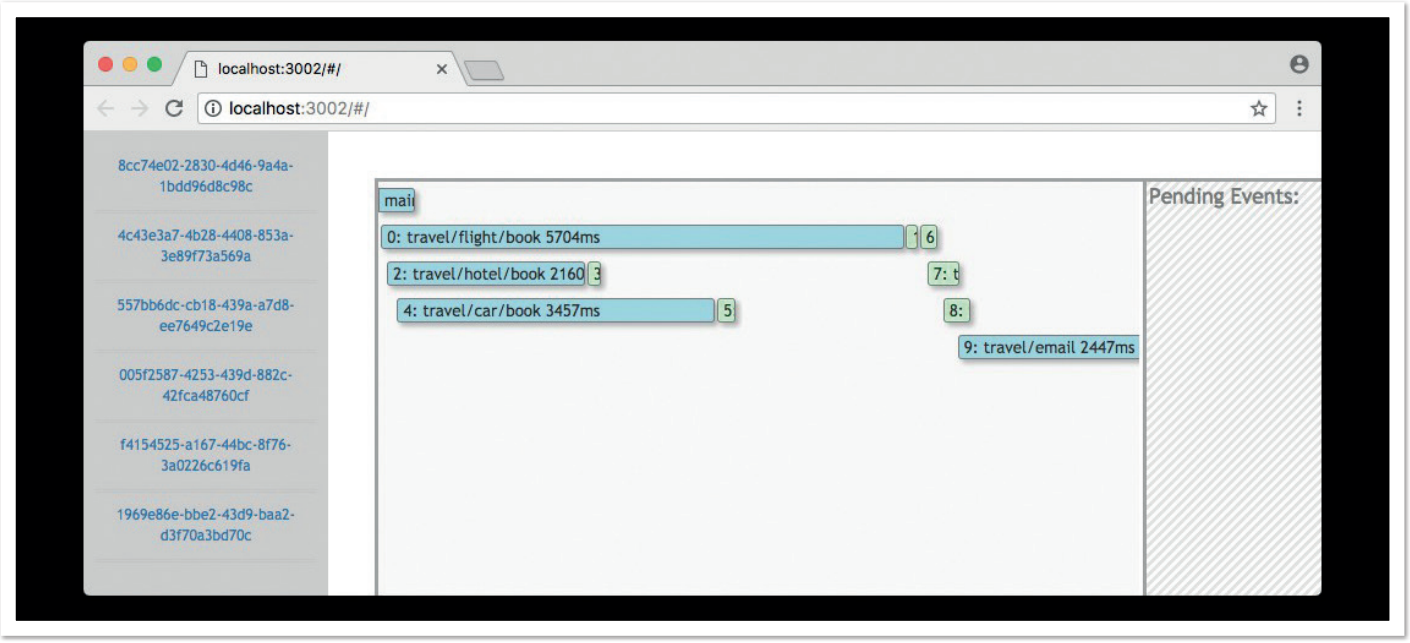

Abbildung 3: Flow-Benutzerinterface - Reisebuchung

tatsächlichen, detaillierten Ablauf inklusive der Ausführungszeiten kann man sich im Flow-Benutzerinterface ansehen. Dieses ist derzeit noch in einem experimentellen Stadium, eignet sich aber hervorragend zur Problem- und Fehler-Analyse. Man findet in der Visualisierung beispielsweise leicht sich blockierende Schritte und Aufrufe. *Abbildung 3* zeigt eine Ausführungs-Instanz der Reisebuchung.

Fn Project in der Produktion

Eine mit Fn erstellte Funktion ist ein Docker-Container. Daher ist eine mit Kubernetes verwaltete Docker-Umgebung eine gute Wahl, um Fn-Funktionen ablaufen zu lassen. Für die einfache Installation der benötigten Fn-Ressourcen wie Fn-Service, Fn-UI, Flow-Service, Flow-UI und Persistenz-Datenbank kann ein vorgefertigtes Helm-Chart genutzt werden. Dabei handelt es sich um einen kommandozeilenbasierten Paketmanager für Kubernetes. Das Helm-Repository mit den für Fn benötigten Konfigurationsdaten kann direkt mit „$ git clone git@github.com:fnproject/fn-helm.git" aus GitHub bezogen werden *[9]*. Danach müssen noch mit „helm dep build fn" die Abhängigkeiten installiert werden. Schließlich installiert man das Chart und setzt mit diesem sowie „helm install --name my-release fn" in einem Arbeitsschritt die Fn-Plattform als verwaltete Docker-Umgebung auf.

Zusammenfassung und Ausblick

Das bestehende Angebot kommerzieller Serverless-Computing- und Functions-as-a-Service-Angebote ist breit. Im Vergleich dazu ist Fn Project spät am Markt. Die kommerziellen Angebote sind allerdings oft proprietär und auf die Nutzung mit den zugehörigen Diensten der Cloud des jeweiligen Anbieters begrenzt. Fn Project geht – auf der langjährigen Erfahrung mit Iron.io *[4]* aufbauend – einen anderen Weg. Es ist vollständig Open Source und konsequent Container-basiert. Dadurch läuft es auf Basis der exakt selben Codebasis *[5]* auf den Laptops der Entwickler, unter Nutzung von Docker-Verwaltungswerkzeugen wie Kubernetes in den eigenen Rechenzentren der Anwender oder in jeder beliebigen öffentlichen Cloud. Zudem ist es einfach erweiterbar und zählt auf die Erfahrung und die Mitwirkung der Entwicklergemeinde.

Während Java – die Programmiersprache mit der auch in Zeiten der Cloud immer noch weitesten Verbreitung *[10]* – von anderen Serverless-Computing-Anbietern stiefmütterlich behandelt wird, startet Fn Project dank des Erfahrungsschatzes der Oracle-Entwickler mit exzellenter Unterstützung. Was zur Vollständigkeit noch fehlt, ist der passende und automatisch skalierende Serverless-Computing-Plattformdienst in der Oracle-Cloud *[8]*.

Weitere Informationen

[1] *aws.amazon.com/lambda*
[2] *cloud.google.com/functions*
[3] *azure.microsoft.com/serverless/functions*
[4] *open.iron.io*
[5] *github.com/fnproject/fn*
[6] *github.com/fnproject/cli/releases*
[7] *fnproject.io/tutorials*
[8] *cloud.oracle.com/tryit*
[9] *github.com/fnproject/fn-helm*
[10] *www.tiobe.com/tiobe-index*

Marcel Amende
marcel.amende@oracle.com

Geboren als Ingenieur, aufgewachsen bei Oracle, zu Hause in der Cloud: Marcel Amende repräsentiert die „Cloud Native"-Entwicklergemeinde bei Oracle, um aus den Diensten der Oracle-Cloud kreative Kundenlösungen zu gestalten. Sein besonderes Interesse gilt dabei den großen Innovationsthemen dieser Tage wie IoT, Blockchain, Mobile & Bots sowie Serverless Computing & Co.