whitehorses

member of the human network

WHITEBOOK

Serverless Java met Fn

September 2018

Auteur:
Mike Heeren
JAVA- EN INTEGRATIESPECIALIST

Inleiding

We zien de laatste tijd veel veranderingen in de opzet van applicaties in de IT wereld. Waar voorheen alles on-premise moest draaien, wordt steeds meer naar de cloud gemigreerd. Ook de oude monolithische opzet van de applicaties, wordt aangepast naar kleine, herbruikbare componenten; Microservices. Met het Fn project kunnen we zelfs nòg een stap kleiner: Functions-as-a-Service (FaaS). Daarnaast kan Fn zowel on-premise als in de cloud draaien.

Serverless? Heb ik dan geen server meer nodig?

Helaas, hoewel de term ‘serverless’ wellicht doet vermoeden dat er geen servers meer nodig zijn om de services of functies op te draaien, is dit niet het geval. Uiteindelijk zullen de functies natuurlijk ergens (op een server) gehost moeten worden.

De term ‘serverless’ komt van het gehanteerde afrekenmodel. De functies zullen uiteindelijk gewoon op een server draaien, maar hiervoor hoeft niet maandelijks hetzelfde bedrag betaald te worden. Je betaalt dus niet voor het beschikbaar zijn van de functies, maar voor de resources die daadwerkelijk gebruikt zijn wanneer de functies wél aangeroepen worden. Daarnaast hoeven ontwikkelaars zich niet druk te maken over de infrastructuur waarop de functies komen te draaien. Of dit nu in de cloud, on-premise, op Windows of op Linux is. De ontwikkelaar kan zich dus puur richten op de functionaliteit die gebouwd moet worden voor eindgebruikers.

Het Fn project

Fn is een open source container-native serverless platform. Zoals eerder genoemd, kan Fn op elk platform draaien. De enige vereiste is dat Docker geïnstalleerd is. Het product zelf is ontwikkeld in Go. Naast Go kunnen echter ook diverse andere programmeertalen gebruikt worden om functies te ontwikkelen, zoals Node.js, Ruby, Python of Java. In dit Whitebook zullen we vooral ingaan op het ontwikkelen van Java functies voor Fn.

Een groot voordeel van Fn is dat requests worden afgehandeld in losse Docker containers, die automatisch worden gestart en gestopt. Hierdoor worden er minder resources gebruikt wanneer functies niet aangeroepen worden.

Het Fn project bestaat uit 4 hoofdonderdelen:

- **Fn Server**
  Dit is de FaaS server waarop de functies gedeployed kunnen worden.
- **Fn LB (Load Balancer)**
  De load balancer om requests te routeren naar de verschillende nodes. Wanneer gebruik wordt gemaakt van *hot functions* (hierover later meer), zorgt de Fn LB ervoor dat verkeer waar mogelijk, gerouteerd wordt naar reeds actieve containers om optimale performance te garanderen.
- **Fn FDK's**
  Het Fn project ontsluit momenteel verschillende *Functions Developer Kits* (FDK's), waarmee functies in verschillende talen kunnen worden ontwikkeld. In dit Whitebook zullen we vooral naar de Java FDK kijken.
- **Fn Flow**
  Fn Flows kunnen worden gebruikt voor orkestratie van workflows. Middels Fn Flows kunnen meerdere functieaanroepen parallel of sequentieel worden uitgevoerd.

Installatie

Nu we wat achtergrondinformatie hebben over wat serverless en Fn is, is het tijd om Fn daadwerkelijk te installeren en de eerste functie te bouwen. Bij het schrijven van het Whitebook is gebruikgemaakt van de volgende applicatieversies:

TYPE	APPLICATIE	VERSIE
OS	Ubuntu	18.04.1
DOCKER	Docker	18.03.1

De installatie van Fn zelf is erg simpel. Op Linux hoeft enkel het volgende commando te worden uitgevoerd:

```
curl -LSs https://raw.githubusercontent.com/fnproject/cli/master/install | sh
```

Nadat de installatie is voltooid, kan de Fn server worden gestart. Hierbij kan de *-d* (of *--detach*) parameter optioneel gebruikt worden om de server op de achtergrond te laten starten. Standaard wordt deze server op poort 8080 gestart, mocht het nodig zijn om een andere poort te gebruiken, kan ook de *-p* parameter gebruikt worden.

```
fn start -d
```

We zien nu dat Docker een *pull* doet van de image, indien deze nog niet beschikbaar is op de server. Vervolgens wordt de container opgestart. Dat de container is gestart, kunnen we zien door het uitvoeren van het onderstaande commando:

```
docker ps
```

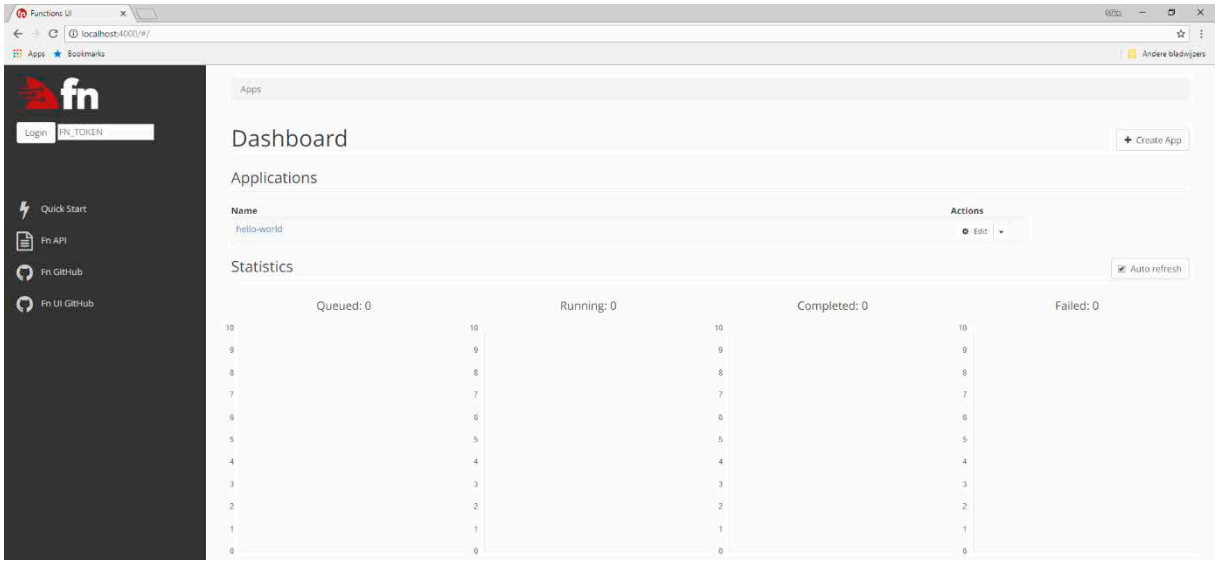
Naast deze container kan ook de UI container worden opgestart. Hiervoor kan het onderstaande commando worden gebruikt. Bij dit commando wordt er een container opgestart met het *fnproject/ui* image, waarbij de *--rm* (*clean up*) parameter wordt gebruikt, zodat de container en het filesystem automatisch worden opgeruimd wanneer de container wordt afgesloten. Via de *--link* parameter wordt het voor de UI container mogelijk om verbinding te maken met de API container. Hiervoor is tevens de *FN_API_URL* omgevingsvariabele nodig. Ook hier wordt de *-d* parameter gebruikt om de container op de achtergrond te laten draaien. De *-p* parameter wordt hier voor Docker port mapping gebruikt. In het onderstaande voorbeeld wordt poort 4000 van de container, aan dezelfde poort van de host machine gekoppeld.

```
docker run --rm --link fnserver:api -d -p 4000:4000 -e "FN_API_URL=http://
api:8080" fnproject/ui
```

Wanneer we nu nogmaals de actieve Docker containers bekijken, zien we dat nu ook de UI container beschikbaar is:

```
root@serverless-java:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
root@serverless-java:~# fn start -d
7b45a20d6b58daf5ea94c0acc665f4f9d835f67be89c0fb86fc52d683cf24525
root@serverless-java:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                              NAMES
7b45a20d6b58        fnproject/fnserver   "./fnserver"        3 seconds ago       Up 2 seconds        2375/tcp, 0.0.0.0:8080->8080/tcp   fnserver
root@serverless-java:~# docker run --rm  --link fnserver:api -d -p 4000:4000 -e "FN_API_URL=http://api:8080" fnproject/ui
168ea54d30b2badb4c8af4ae47933b1a5c8c5f23bc96e8979353a033aeaa8989
root@serverless-java:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                              NAMES
168ea54d30b2        fnproject/ui        "npm start"         6 seconds ago       Up 6 seconds        0.0.0.0:4000->4000/tcp             angry_rosalind
7b45a20d6b58        fnproject/fnserver   "./fnserver"        20 seconds ago      Up 19 seconds       2375/tcp, 0.0.0.0:8080->8080/tcp   fnserver
root@serverless-java:~#
```

De UI console is vervolgens ook direct bereikbaar via http://localhost:4000.



Hello, World!

Nu de server draait, is het tijd om een eerste applicatie/functie te bouwen. Hiervoor kan de *Fn Java FDK (Functions Developer Kit)* gebruikt worden. We kunnen de volledige boilerplate code voor een functie genereren, door het volgende commando uit te voeren. Let er hierbij op dat de applicatie in de huidige directory wordt gegenereerd.

```
fn init --runtime java hello-world
```

Binnen de directory */hello-world* is nu een viertal bestanden gegenereerd:

- **func.yaml**
  Dit is een YAML bestand met configuratie van de functie. Zo is hier onder andere te zien dat de functienaam *hello-world* is en de versie 0.0.1. Ook zien we dat de *handleRequest* methode uit de (gegenereerde) *HelloFunction* klasse wordt gebruikt voor het afhandelen van de functieaanroepen.
- **pom.xml**
  Dit bestand wordt gebruikt voor het bouwen en testen van de functie.
- **src/main/java/com/example/fn/HelloFunction.java**
  Gegenereerde klasse, waarin ook direct een implementatie van de *handleRequest* methode (uit het *func.yaml* bestand) is gegenereerd.
- **src/test/java/com/example/fn/HelloFunctionTest.java**
  Unittest om de *HelloFunction* klasse te testen via JUnit.

Als we de functie willen starten, moeten we eerst het volgende commando uitvoeren. Dit commando zorgt ervoor dat we de functies lokaal kunnen draaien, zonder ze naar een registry als Docker Hub te pushen.

```
export FN_REGISTRY=fndemouser
```

Het is nu mogelijk om de functie aan te roepen. We doen eerst een aanroep zonder parameters, en vervolgens een waar we een *String* (de *input* parameter van de *handleRequest* methode) aan meegeven. Beide commando's moeten worden uitgevoerd vanuit de */hello-world* directory.

```
fn run
echo -n "Mike" | fn run
```

```
root@serverless-java:/home/whitehorses/workspace/fnproject/hello-world# fn run
Building image hello-world:0.0.1
Hello, world!
root@serverless-java:/home/whitehorses/workspace/fnproject/hello-world# echo -n "Mike" | fn run
Building image hello-world:0.0.1
Hello, Mike!
root@serverless-java:/home/whitehorses/workspace/fnproject/hello-world#
```

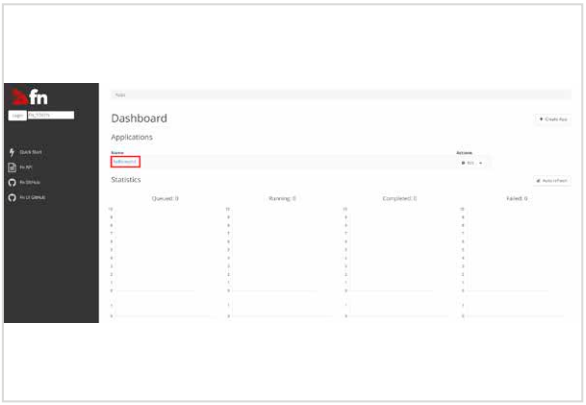
Dit commando zorgt ervoor dat er een Docker container wordt opgestart. Deze container verwerkt vervolgens het request. Nadat het request verwerkt is, wordt de container ook direct weer afgesloten:

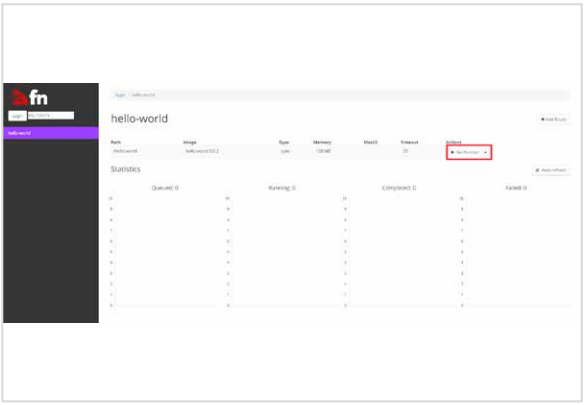
```
root@serverless-java:~# docker ps
CONTAINER ID        IMAGE                COMMAND                  CREATED             STATUS                  PORTS                              NAMES
ec179d0e31c1        hello-world:0.0.1    "/usr/bin/java -XX:+…"   1 second ago        Up Less than a second                                      loving_perlman
506b9614f5fc        fnproject/ui         "npm start"              3 minutes ago       Up 3 minutes            0.0.0.0:4000->4000/tcp             optimistic_lovelace
665122f42e6d        fnproject/fnserver   "./fnserver"             3 minutes ago       Up 3 minutes            2375/tcp, 0.0.0.0:8080->8080/tcp   fnserver
root@serverless-java:~# docker ps
CONTAINER ID        IMAGE                COMMAND                  CREATED             STATUS              PORTS                              NAMES
506b9614f5fc        fnproject/ui         "npm start"              3 minutes ago       Up 3 minutes        0.0.0.0:4000->4000/tcp             optimistic_lovelace
665122f42e6d        fnproject/fnserver   "./fnserver"             4 minutes ago       Up 4 minutes        2375/tcp, 0.0.0.0:8080->8080/tcp   fnserver
root@serverless-java:~# docker ps
CONTAINER ID        IMAGE                COMMAND                  CREATED             STATUS                  PORTS                              NAMES
d36bdf3745cd        hello-world:0.0.1    "/usr/bin/java -XX:+…"   1 second ago        Up Less than a second                                      modest_leakey
506b9614f5fc        fnproject/ui         "npm start"              3 minutes ago       Up 3 minutes            0.0.0.0:4000->4000/tcp             optimistic_lovelace
665122f42e6d        fnproject/fnserver   "./fnserver"             4 minutes ago       Up 4 minutes            2375/tcp, 0.0.0.0:8080->8080/tcp   fnserver
root@serverless-java:~# docker ps
CONTAINER ID        IMAGE                COMMAND                  CREATED             STATUS              PORTS                              NAMES
506b9614f5fc        fnproject/ui         "npm start"              3 minutes ago       Up 3 minutes        0.0.0.0:4000->4000/tcp             optimistic_lovelace
665122f42e6d        fnproject/fnserver   "./fnserver"             4 minutes ago       Up 4 minutes        2375/tcp, 0.0.0.0:8080->8080/tcp   fnserver
root@serverless-java:~#
```

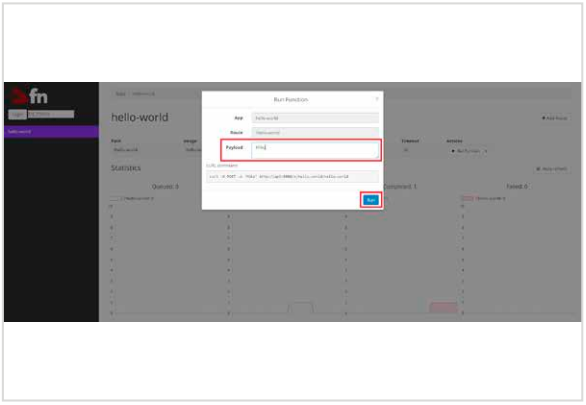
In plaats van de functie slechts één keer uit te voeren, is het ook mogelijk om deze te deployen naar de Fn server. Dit kan worden gedaan middels het onderstaande commando. Bij het uitvoeren van dit commando, wordt ook automatisch het versienummer in het *func.yaml* bestand opgehoogd.

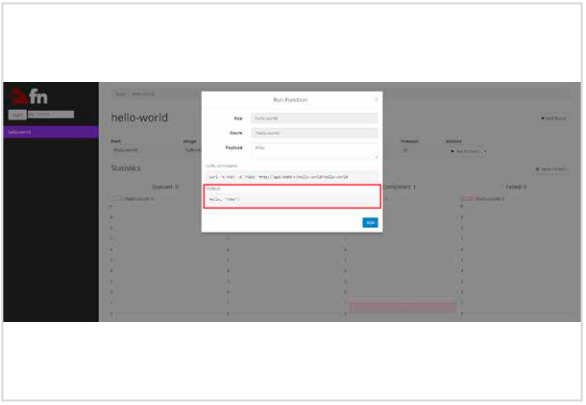
```
fn deploy --local --app hello-world
```

Wanneer we nu nogmaals in de UI console kijken, zien we ook dat deze applicatie gedeployed is. Als je doorklikt naar de applicatie, kan de functie tevens aangeroepen worden via *Run Function*. Hier kan optioneel een payload worden ingevoerd. Na het uitvoeren van de functie wordt het resultaat direct getoond.









Tevens wordt het cURL commando getoond dat onderwater uitgevoerd wordt. In dit commando is de URL van de functie te zien. Houd er wel rekening mee dat http://api, vervangen moet worden door http://localhost wanneer je dit commando zelf uit wilt voeren.

Cold functions vs. Hot functions

Er zijn twee typen functies die op de Fn server gedraaid kunnen worden. Standaard is een functie gespecificeerd als een ‘cold’ function. Dit betekent dat voor ieder request een nieuwe container wordt gestart om het request af te handelen. Nadat het request is verwerkt, wordt deze container direct weer afgesloten. Dit is dus het gedrag wat we ook constateren bij het uitvoeren van het ‘fn run’ commando.

Er bestaan echter ook ‘hot’ functions. Hierbij wordt er een container opgestart, die vervolgens een bepaalde tijd (dit is standaard 30 seconden, maar aan te passen via de timeout setting in het func.yaml bestand) actief blijft. Wanneer er binnen deze time-out geen verdere requests zijn ontvangen door de container, zal de container alsnog afgesloten worden. Voor de performance van de requests, zijn ‘hot’ functions logischerwijs een stuk gunstiger. Hiervoor hoeft namelijk niet voor ieder request een nieuwe container gestart te worden. Volgens de documentatie van Fn, moet er vanuit worden gegaan dat het starten van de container gemiddeld 300ms kost.

Na het aanroepen van de hello-world functie via de UI console valt één ding meteen op bij het ophalen van de actieve Docker processen. Hoewel in de documentatie staat dat een functie standaard ‘cold’ is, zien we toch echt dat het request als hot function wordt verwerkt:

```
root@serverless-java:~# docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS                    PORTS                              NAMES
c591573edbfc        hello-world:0.0.2      "/usr/bin/java -XX:+…"   30 seconds ago      Up 30 seconds (Paused)                                       01CP05MZ0
BNG8G00GZJ000000E
168ea54d30b2        fnproject/ui           "npm start"              3 hours ago         Up 3 hours                0.0.0.0:4000->4000/tcp             angry_ros
alind
7b45a20d6b58        fnproject/fnserver     "./fnserver"             3 hours ago         Up 3 hours                2375/tcp, 0.0.0.0:8080->8080/tcp   fnserver
```

Dit is te verklaren door het gegenereerde func.yaml bestand. Hierin is het format op http ingesteld. Dit is een van de formaten die wordt ondersteund door hot functions. Wanneer we het format aanpassen naar een formaat dat geen hot functions ondersteunt (zoals default), zien we bij meerdere aanroepen ook het verwachte gedrag van een cold function weer terug:

```
root@serverless-java:~# docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS              PORTS                              NAMES
9339f2520f37        hello-world:0.0.3      "/usr/bin/java -XX:+…"   6 seconds ago       Up 2 seconds                                           01CP08PC73NG8
00GZJ000001E
8a5e9bb6a76b        hello-world:0.0.3      "/usr/bin/java -XX:+…"   6 seconds ago       Up 2 seconds                                           01CP08PC4MNG8
00GZJ000001C
d9049420d4c1        hello-world:0.0.3      "/usr/bin/java -XX:+…"   6 seconds ago       Up 2 seconds                                           01CP08PC2ENG8
00GZJ000001A
92db1ef83351        hello-world:0.0.3      "/usr/bin/java -XX:+…"   6 seconds ago       Up 2 seconds                                           01CP08PC1FNG8
00GZJ0000016
cd32bac03e02        hello-world:0.0.3      "/usr/bin/java -XX:+…"   6 seconds ago       Up 2 seconds                                           01CP08PC11NG8
00GZJ0000012
3828ac199fb5        hello-world:0.0.3      "/usr/bin/java -XX:+…"   6 seconds ago       Up 3 seconds                                           01CP08PC1GNG8
00GZJ0000018
d47a00ad0d79        hello-world:0.0.3      "/usr/bin/java -XX:+…"   6 seconds ago       Up 3 seconds                                           01CP08PC17NG8
00GZJ0000014
4e333df315c1        hello-world:0.0.3      "/usr/bin/java -XX:+…"   6 seconds ago       Up 3 seconds                                           01CP08PBZ0NG8
00GZJ0000010
168ea54d30b2        fnproject/ui           "npm start"              4 hours ago         Up 4 hours          0.0.0.0:4000->4000/tcp             angry_rosalind
7b45a20d6b58        fnproject/fnserver     "./fnserver"             4 hours ago         Up 4 hours          2375/tcp, 0.0.0.0:8080->8080/tcp   fnserver
```

Fn Flow

Met Fn Flow kunnen langer lopende processen, inclusief orkestratie, worden opgebouwd. Het opbouwen van een flow kan momenteel enkel nog gedaan worden met de Java FDK. In de toekomst zou het echter ook mogelijk moeten worden om deze flows in JavaScript, Python of Go te ontwikkelen.

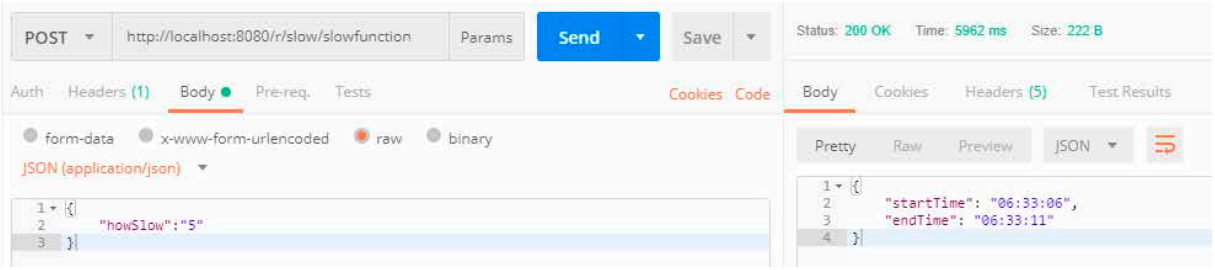
Om Fn Flows te kunnen gebruiken moet, naast de reguliere Fn Server, ook de Flow Service gestart worden. Dit kan worden gedaan middels de volgende commando's:

```
DOCKER_LOCALHOST=$(docker inspect --type container -f '{{.NetworkSettings.
Gateway}}' fnserver)
docker run --rm -p 8081:8081 -d -e API_URL="http://$DOCKER_
LOCALHOST:8080/r" -e no_proxy=$DOCKER_LOCALHOST --name flow-service
fnproject/flow:latest
```

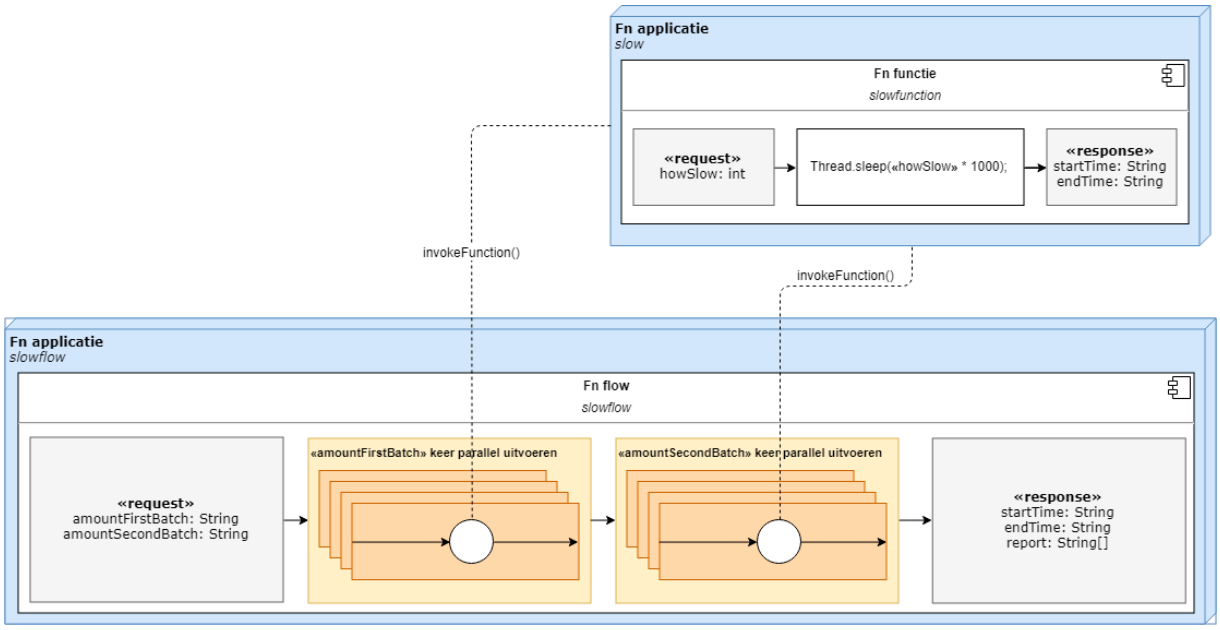
Naast de Flow Service, is het ook mogelijk om een UI service te starten, waarmee realtime kan worden bekeken wat er in de flows gebeurt. Voor het starten van de UI service, moet het onderstaande worden uitgevoerd:

```
DOCKER_LOCALHOST=$(docker inspect --type container -f '{{.NetworkSettings.
Gateway}}' fnserver)
docker run --rm -p 3000:3000 -d --name flowui -e API_URL=http://$DOCKER_
LOCALHOST:8080 -e COMPLETER_BASE_URL=http://$DOCKER_LOCALHOST:8081
fnproject/flow:ui
```

Zoals eerder al genoemd kan een Fn Flow ontwikkeld worden middels dezelfde Java FDK, waar ook reguliere Fn functies in kunnen worden ontwikkeld. Om dit te testen heb ik een eenvoudige Fn functie ontwikkeld en gedeployed, waaraan een JSON object kan worden meegegeven. In dit JSON object is een tijd in te stellen die moet worden gewacht voordat de functie een resultaat teruggeeft. Het resultaat van de service bestaat vervolgens uit de begin- en eindtijd van de executie:



Deze functie gaan we nu vanuit een Fn Flow meerdere malen, in twee batches, parallel aanroepen. De grootte van deze batches kan worden opgegeven bij het aanroepen van de Fn Flow. Onderstaand een schematisch overzicht van deze opzet.



Dit is met een Fn Flow betrekkelijk eenvoudig te realiseren door een nieuwe Fn functie te genereren op dezelfde manier als we eerder al hadden gedaan. Vervolgens passen we de implementatie aan naar iets als het volgende:

```
public SlowFlowResponse handleRequest(SlowFlowRequest request) {
    // Initiele informatie voor de response ophalen
    SimpleDateFormat timeFormat = new SimpleDateFormat("HH:mm:ss");
    String startTime = timeFormat.format(new Date());
    // Opbouwen eerste batch. Grootte van de batch wordt bepaald door het request.
    FlowFuture[] firstBatch = new FlowFuture[request.getAmountFirstBatch()];
    for (int i = 0; i < request.getAmountFirstBatch(); i++) {
        SlowFunctionRequest batchReq = new SlowFunctionRequest();
        batchReq.setHowSlow(2);
        // "slowfunction" uit applicatie "slow" aanroepen met opgebouwd request.
        firstBatch[i] =currentFlow().invokeFunction("slow/
slowfunction", batchReq, SlowFunctionResponse.class);
    }
    // Het daadwerkelijk uitvoeren van de eerste batch
    currentFlow().allOf(firstBatch).get();
    // Opbouwen tweede batch. Wederom wordt de grootte bepaald door het request.
    FlowFuture[] secondBatch = new FlowFuture[request.getAmountSecondBatch()];
```

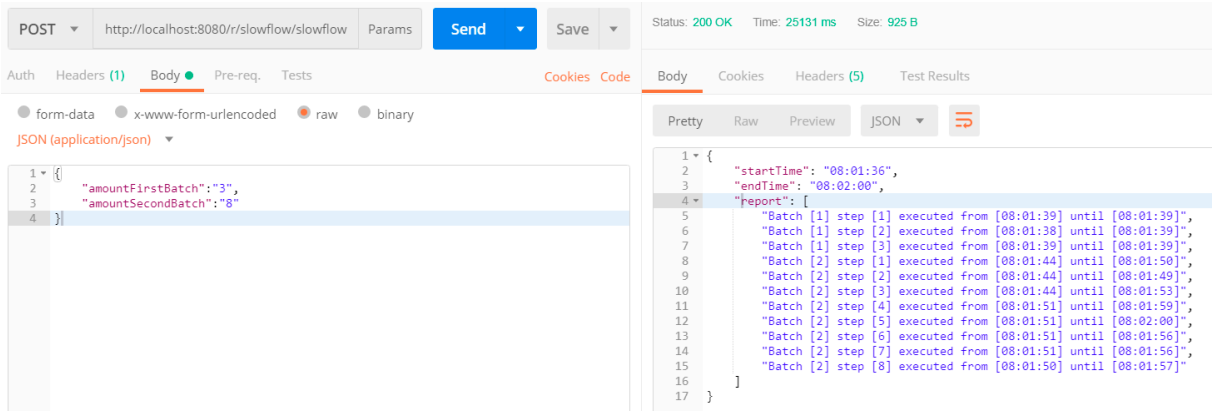
```
    for(int i = 0; i < request.getAmountSecondBatch(); i++) {
        SlowFunctionRequest batchReq = new SlowFunctionRequest();
        batchReq.setHowSlow(3);
        // "slowfunction" uit applicatie "slow" aanroepen met opgebouwd request.
        secondBatch[i] = currentFlow().invokeFunction("slow/
slowfunction", batchReq, SlowFunctionResponse.class);
    }
    // Het daadwerkelijk uitvoeren van de tweede batch
    currentFlow().allOf(secondBatch).get();
    // Opbouwen van het response.
    String[] report = new String[firstBatch.length + secondBatch.length];
    for (int i = 0; i < firstBatch.length; i++) {
        SlowFunctionResponse stepRes = (SlowFunctionResponse) firstBatch[i].get();
        report[i] = "Batch [1] step [" + (i + 1) + "] executed from [" + stepRes.
getStartTime() + "] until [" + stepRes.getEndTime() + "]";
    }
    for (int i = 0; i < secondBatch.length; i++) {
        SlowFunctionResponse stepRes = (SlowFunctionResponse) secondBatch[i].
get();
        report[firstBatch.
length + i] = "Batch [2] step [" + (i + 1) + "] executed from [" + stepRes.
getStartTime() + "] until [" + stepRes.getEndTime() + "]";
    }
    String endTime = timeFormat.format(new Date());
    return new SlowFlowResponse(startTime, endTime, report);
}
```

Ook het deployen en starten van een Fn Flow kan op dezelfde manier worden gedaan als bij een reguliere Fn functie. Nadat de flow gedeployed is, moet alleen nog een extra parameter aan de applicatie worden toegevoegd, zodat deze weet hoe de Flow Service te bereiken is. Dit kan via de UI console, of via het onderstaande commando worden gedaan:

```
fn config app flowflow COMPLETER_BASE_URL "http://$DOCKER_LOCALHOST:8081"
```

Wanneer de functie gedeployed is en de *COMPLETER_BASE_URL* parameter ingesteld, kan bijvoorbeeld de volgende aanroep gedaan worden.

Let op: In tegenstelling tot de voorbeeldimplementatie hierboven, wordt in het onderstaande voorbeeld geen gebruikgemaakt van statische waarden bij het aanroepen van de *slowfunction*. Hiervoor worden willekeurige waarden tussen de 1 en de 10 gebruikt.



Wanneer we nu de Fn Flow console (dit is dus de console die op poort 3000 is gestart, en niet de standaard UI console op poort 4000!) openen, kunnen we ook hier de verwerking van de flow zien.

Conclusie

Het ontwikkelen van Functions(-as-a-Service) via Fn werkt erg goed. Een groot voordeel is dat zowel on-premise, in de cloud als op het systeem van een ontwikkelaar zelf snel functies gedraaid- en/of getest kunnen worden. Daarnaast worden er minder resources gebruikt wanneer er geen functies worden aangeroepen. Dit scheelt weer in de kosten.

Wanneer functies erg vaak worden aangeroepen en hoge performance een absolute *must* is, kan het ongunstig zijn om voor ieder request een nieuwe Docker container op te moeten starten. Het opstarten van deze container kost namelijk ook tijd. Door de introductie van *hot functions* tackled Fn echter direct een groot deel van dit probleem.

Orkestratie van (langer lopende) processen via Fn Flow lijkt daarnaast ook goed te werken. Hoewel de orkestratie van deze processen hier in code moet worden gedaan, in plaats van grafische interfaces zoals bij integratie platforms van bijvoorbeeld Oracle of MuleSoft, lijkt de learning curve mee te vallen. De ontwikkeling van Fn Flows gebeurt namelijk op exact dezelfde wijze als de ontwikkeling van reguliere Fn functies.

Bronnen

FnProject.io
GitHub: Fn
GitHub: Fn - Frequently Asked Questions
GitHub: Fn - Hot functions
GitHub: Fn Load Balancer
GitHub: Fn Java Functions Developer Kit (FDK)
GitHub: Fn Flow
GitHub: Fn Flow UI