Imparare Kubernetes con OpenWhisk: creare l’immagine docker

Apache OpenWhisk è un sistema Serverless Open Source molto flessibile. È una tipica applicazione distribuita e puo quindi essere installato in Kubernetes come sistema distribuito. Utilizzare Kafka come coda di messaggi e CouchDB come database per mantenere configurazioni e attivazioni.

Tutti i componenti sono replicabili e scalabili e si può creare un ambiente con centinaia di nodi. Per questo motivo è un ambiente ideale da installare per apprendere come funziona Kubernetes in ambienti realisitici. Ho deciso pertanto di scrivere una serie di articolo su come si installa OpenWhisk in Kubernetes allo scopo di presenare un tutorial su Kubernetes.

Non copriró in questa serie invece come funziona OpenWhisk stesso come sistema serverless, anche se è indubbiamente un argomento anch'esso interessante. Delego a questo proposito la lettura del libro in inglese "Learning Apache OpenWhisk", pubblicato da O'Reilly, di cui sono l'autore. Cortesemente Google ne pubblica un ampio estratto di ben 94 pagine che copre tutte le basi, lo trovate facilmente su Google stesso con la chiave"Learning Apache OpenWhisk". Inoltre c'è su YouTube in italiano un corso gratuito che copre Nimbella, che è un servizio commerciale Serveless basato su OpenWhisk. La playlist ha URl bit.ly/intronim

# Compilazione in locale

Iniziamo le operazioni di installazione partendo dalla modalità più semplice, installazione in locale in modalità singola ("standalone"). Questo passo è propedeutico per poi creare una immagine Docker da usare in Kubernetes.

Quindi si può cominciare ad usarlo in maniera molto semplice ed immediata. Tutto quello che vi serve è Docker, un Java Development Kit per compilare il backend, e Node.js per poter compilare l front-end. Se siete uno sviluppatore molto probabilmente avete già tutto questo nella vostra macchina.

Se avete i prerequisiti sulla vostra macchina potete cimentarvi nell'impresa, tutto sommato piuttosto semplice, di installarlo e farlo partire in locale. Potete semplicemente scaricare i sorgenti da qui:

<https://github.com/apache/openwhisk>

Nelle releases troverete la versione 1.0, in formato zip e tar.gz. Vi conviene comunque prendere il master con un bel git clone, in quando è stabile e di solito più aggiornato delle releases, e quindi contiene le ultime novità e bug fix.

Una volta scaricato e spacchettato, o clonato, andate nella directory principale e potete compilare OpenWhisk standalone con

./gradlew :core:standalone:build

Dovete avere un attimo di pazienza (ci mette un po' a compilare) e avrete OpenWhisk versione locale pronto all'uso. Mentre compila vi consiglio anche di andare su:

<https://github.com/apache/openwhisk-cli>

per scaricare la command line interface per il vostro sistema operativo. Una volta scaricata, dezippate il pacchetto e mettete il comandu **wsk** nel *PATH* da qualche parte nel vostro sistema (per esempio tipicamente in  */usr/local/bin* su Mac e Linux, o *C:\Windows\System32* su Windows).

Tutto è pronto per lanciarvi nel mondo del serverless! Potete lanciare OpenWhisk in locale con il comando:

java –jar bin/openwhisk-standalone.jar

Questo dovrebbe normalmente aprire il browser con il playground. Si tratta di una semplice applicazione javascript che vi permette di scrivere una funzione Serverless nei vari linguaggi direttamente nel browser e provarla.

# Sperimentare con il serverless

Una volta che avete fatto hello world nel playground, la cosa migliore è scrivere delle applicazioni con un editor, visto che il playground non è un completo ambiente di sviluppo. E qui viene utile la utility a riga di comando **wsk**.

La cli la dovete innanzitutto configurare. Se leggete accuratamente l'output di OpenWhisk quando lo lanciate (fate attenzione, è verso l'inizio, e viene immediatamente 'scrollato via' dagli altri messaggi che vengono mostrati) troverete il comando per configurare.

A questo punto copiate uno degli esempi che vi presenta il playground, per esempio *hello.js* in un file, e scrivete:

wks action update hello hello.js

per creare l'azione hello. A questo punto la potete invocare e vedere l'output con

wsk action invoke hello –p name Michele -r

per vedere l'output.

# Creare l'immagine Docker di OpenWhisk

Adesso che sappiamo come compilare OpenWhisk in locale è il momento di automatizzare il tutto scrivendo un *Dockerfile.*

Un Dockerfile è un descrittore che Docker utilizza per creare le sue immagini e contiene generalmente tutti i comandi di compilazione necessari più eventuali direttive sempre relativa alla creazione di immagini. Vediamo passo passo come scrivere un Dockerfile per OpenWhisk standalone

La prima direttiva che vedremo è *FROM* che sostanzialmente indica una immagine base che possiamo usare come punto di partenza. Poiché OpenWhisk è fatto in Scala iniziamo proprio con

FROM scala as builder

WORKDIR

Notare che abbiamo dato un nome all'immagine, perché la utilizzeremo solo come area temporanea per la compilazione e ci serve il nome per riferirla dopo. E notare che abbiamo impostato come directory principale la radice.

A questo punto ci preoccupiamo di aggiungere un po' di componenti che si serviranno per la compilazione. Java (che è usato da scala) è già presente nell'immagine scala, ma mancano *git* (che serve per scaricare i sorgenti) e *npm* più *nodejs* (che servono per compilare il frontend) , quindi li aggiungiamo usando la direttiva *RUN:*

RUN apt-get update && apt-get install -y git nodejs npm

L'immagine di scala è basata, come moltissime altre immagini docker, su Linux Debian. Quindi utilizziamo *apt-get install* che è il comando di questa distribuzione per installare pacchetti. Notare che prima di installare dobbiamo precedere il comando con una *apt-get* *update* per aggiornare gli indici per scaricare i pacchetti.

Ci servono anche degli altri componenti che non sono disponibili nel repository ufficiale di Debian. In particolare ci serve una specifica versione di Docker compilata in modalità statica e la riga di comando di OpenWhisk. Aggiungiamo questi componenti con dei download diretti dei pacchetti binari in questione:

RUN curl [https://download.docker.com/linux/static/stable/x86\_64/docker-18.06.3-ce.tgz](https://download.docker.com/linux/static/stable/aarch64/docker-18.06.3-ce.tgz) | tar xzvf -

RUN curl -sL <https://github.com/apache/openwhisk-cli/releases/download/1.2.0/OpenWhisk_CLI-1.2.0-linux-amd64.tgz> | tar xzvf -

A questo punto possiamo scaricare e compilare: "cloniamo" i sorgenti da GitHub, cambiamo alla directory di lavoro e compiliamo il tutto:

RUN git clone <https://github.com/apache/openwhisk>

RUN cd openwhisk && ./gradlew :core:standalone:build

Ora l'immagine è pìu o meno pronta, ma non vogliamo portarci dietro tutti i sorgenti e gli artefatti della compilazione. Per questo motivo usiamo la feature di Docker che permette di fare delle build, e poi usare una altra immagine come base per produrre il risultato finale, estraendo dalla prima immagine i file che servono.

Quindi facciamo esattamente questo: aggiungiamo al Dockerfile le direttive per "ripartire" da capo con l'immagine scala, aggiorniamo i pacchetti all'ultima versione e poi ci mettiamo dentro solo quello che serve.

FROM scala

RUN apt-get update && apt-get -y upgrade

COPY --from=builder /openwhisk/bin/openwhisk-standalone.jar /lib/openwhisk-standalone.jar

COPY --from=builder /docker/docker /usr/bin/docker

COPY --from=builder /wsk /usr/bin/wsk

COPY start.sh /usr/local/bin/start.sh

Tutto pronto per "rilasciare". Impostiamo ora la directory principale (usiamo */home*) ed il comando di avvio (spiegato nel prossimo paragrafo).DDe

ADD start.sh /usr/bin/start.sh

WORKDIR /homem

ENTRYPOINT ["/usr/bin/bash", "/usr/bin/start.sh"]

È importante notare che utilizzando *ENTRYPOINT* questo comando viene sempre eseguito quando lanceremo l'immagine con *docker run <immagine> [<parametri>]*. I parametri, come vedremo, verranno propagati al runtime stesso.

## Lo script di avvio

OpenWhisk ha parecchi parametri, che useremo intensamente nei prossimi articoli. Questi parametri possono essere passati con le system property di Java , con le variabili di ambiente o con file di configurazione. AI nostri fini, per l'esecuzione in una singola immagine docker occorre settare 3 properties:

* Il nome dell'host all'interno dell'immagine docker
* L'indirizzo IP dove il runtime deve stare in ascolto
* L'indirizzo esterno che si usa per connettersi

I primi due posono essere ricavati da un comando Linux: $(hostname) e $(hostname –i). Il terzo dipende da una configurazione e quindi va passata con una variabile di ambiente.

Infine, dato che all'inerno di una immagine Docker non possiamo lanciare il browser, dovremo chiedere a OpenWhisk di farlo, con il parametro –no-browser. Tradotto in codice questo diventa il seguente scritpt *start.sh*:

java $EXTRA\_JVM\_ARGS\

-Dwhisk.standalone.host.name="$(hostname)"\

-Dwhisk.standalone.host.internal="$(hostname -i)"\

-Dwhisk.standalone.host.external="$HOST\_EXTERNAL" \

-jar /usr/lib/openwhisk-standalone.jar --no-browser "$@"

# Lanciare l'immagine Docker

Siamo pronti per lanciare la nostra immagine Docker, ma dobbiamo ricordarci subito che generalmente le immagini richiedono un certo numero di parametri per funzionare

Quindi se facciamo semplicemente:

docker run sciabarracom/openwhisk-standalone:2020-07-01

Non funzionerà! Ecco una breve lista di informazioni che dobbiamo passare all'immagine (e che saranno essenziali per scrivere la configurazione per eseguirlo su Kubernetes):

* Dare un nome e un hostname all’immagine
* Specificare Reti e Porte da utilizzare
* Montare volumi
* Specificare Variabili di ambiente
* Passare Argomenti di riga di comando

Utilizzeremo queste informazioni per scrivere uno script di avvio.

## Diamo un nome al nostro container

Quando lanciamo una immagine, con il comando *docker run* viene creato un container a cui viene dato un nome generato casualmente tipo *nice\_nightingale* o giù di lí. Per quanto pittoresco, questo nome di solito non è molto utile quindi è meglio poterlo specificare direttamente.

Infine è opportuno dare un "hostname" al container, in quanto questo nome viene utilizzato per "riferirlo" da altri container, usando lo switch *-*

Quindi utilizzeremo qualcosa come

--name openwhisk –h openwhisk --rm

Notare che abbiamo aggiunto il flag –*rm,* che serve per rimuovere il container quando viene stoppato. Questo perché OpenWhisk non ha bisogno di preservare nulla (userei un volume in quel caso) ma se provo a lanciarlo nuovamente specificando un nome fisso otterrei errore in quanto esiste già (dovrei invece ristartarlo in quel caso).

## Impostare la porta e il network

Cominciamo dal network e dalle porte esposte.

Prima di tutto occorre sapere che ogni container è eseguito in un network "virtuale" che non è lo stesso in cui gli altri programmi del vostro computer sono eseguiti. Questo fatto per motivi di protezione e isolamento. Per questo motivo se volete accedere al container dovete attaccarlo al network usato dal vostro computer. Questo network si chiama *bridge*, è appunto un bridge in senso TCP/IP che collega il network della vostra macchina al network interno di Docker.

Poi OpenWhisk standalone richiede due porte TCP: 3232 e 3233. La prima viene usata accedervi con il comando *wsk* con una API, e la seconda per il playground

Quindi occorre specificare, dopo *run* e prima del nome dell'immagine i parametri:

--network bridge -p 3822:3288 –p 3289:3828

Se ci dimentichiamo di specificare questi parametri semplicemente non potremo accedere al servizio che è eseguito dentro il container

## Montare i volumi

Nel primo passo abbiamo creato una immagine che è paragonabile al disco iniziale di una macchina virtuale. Quando viene lanciata, si crea un container, che è un filesystem temporaneo che contiene l'immagine ma è scrivibile. L'applicazione dentro il container può utilizzare questo filesystem. Però è un filesystem temporaneo che viene immediatamente distrutto quando il container viene eliminato.

Se si vogliono preservare dei dati (per esempio i file di database) occorre fornire all'immagine un filesystem non transiente che viene preservato. Questo è un volume. Si utilizza il parametro –v per "montare" un volume in un container.

I volumi sono utilizzati anche per montare directory o singoli file, anche speciali (come i socket).

Nel nostro caso, OpenWhisk standalone non ha bisogno di volumi ma ha bisogno di leggere e scrivere in un file speciale: il socket di comunicazione con Docker. Per questo motivo dovremo montarlo come volume.

Infatti se lanciamo la nostra immagine senza questo parametro otteniamo subito un errore:

/usr/bin/docker: Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?.

La soluzione è aggiungere il parametro:

-v /var/run/docker.sock:/var/run/docker.sock

**Attenzione**: se lanciate il comando da windows dovete raddoppiare il primo slash (*-v //var*)

## Variabili di Ambiente

Un altro importante aspetto da considerare sono le variabili di ambiente che sono il modo privilegiato con cui si passano parametri alle applicazioni sono le variabili di ambiente. In particolare qui dobbiamo specificare la variabile di ambiente *HOST\_EXTERNAL* che è la variabile usata per indicare il nome dell'host pubblico. Nel nostro caso è localhost (lo useremo solo localmente per ora) quindi il pametro da aggiungere è:

-e HOST\_EXTERNAL=localhost

## Lo script di avvio

Riepilogando tutto quanto detto prima lo script id avvio è il seguente:

#!/bin/bash

docker run -ti \

-h openwhisk --name openwhisk --rm \

--network bridge -p 3232:3232 -p 3233:3233 \

-v /var/run/docker.sock:/var/run/docker.sock \

-e HOST\_EXTERNAL=localhost \

sciabarracom/openwhisk-standalone:2020-07-01 "$@"

# Conclusioni

Abbiamo finito la prima revisione del nostro Dockerfile... quindi proviamolo, compilando il tutto. Come tag gli diamo la data di oggi:

docker build . -t sciabarracom/openwhisk-standalone:2020-07-01

Ci mette un po' ma alla fine abbiamo la nostra immagine docker !

Potete risparmiarvi la fatica di fare una build semplicemente utilizzando l’immagine che ho pubblicato per questo articolo come: *sciabarracom/openwhisk-standalone:2020-07-01.*