Apache OpenWhisk

Uriel Piñeyro

Teoría detrás de esta tecnología

1 ¿Qué es Apache OpenWhisk?

Es una plataforma de código abierto, distribuída y sin servidor que ejecuta funciones (fx) en respuesta a eventos en cualquier escala. OpenWhisk se encarga de administrar la infraestructura, servidores y escalado usando contenedores de Docker, para que así el desarrollador solo deba enfocarse en crear aplicaciones.

1.1 ¿Qué es la computación sin servidor?

Es un modelo de ejecución basado en la nube, en la que un proveedor de servicios cloud (como AWS, Microsoft Azure, Google Cloud Platform, etc) se encarga de correr el servidor y administrar dinámicamente los recursos que se utilizan.

La ventaja de este tipo de computación es que el despliegue de aplicaciones puede ser mucho más sencilla para el programador, ya que el escalado, la planificación de capacidad y las tareas de mantenimiento pueden estar ocultas para ella o él, y que de eso se encargue el proveedor.

1.2 Acerca de Apache OpenWhisk

Su modelo de programación es como sigue: https://openwhisk.apache.org/images/illustrations/OW-Programming-Model-Horz.png

Y, como se puede ver en la imagen, es **dirigida por eventos**, es decir, que la ejecución no es secuencial, sino que responde a los sucesos generados por el usuario, el sistema o el mismo programador.

Estos eventos dirigen la ejecución sin servidor (serverless execution) del codigo "funcional" llamado **Acciones**. Estos eventos pueden provenir de diversas fuentes como ser:

- Bases de Datos (actualizaciones)
- Colas de Mensajes
- Aplicaciones web y mobile
- Sensores
- ChatBots
- Tareas programadas (mediante alarmas)
- etc.

1.3 Acciones

Son funciones *sin estado* que se ejecutan en la plataforma OpenWhisk. Estas acciones encapsulan la lógica que será ejecutada en respuesta a eventos. Pueden ser invocadas de manera manual vía la OpenWhisk REST API, OpenWhisk CLI, APIs simples creadas por el usuario o automatizadas vía disparadores.

Por ejemplo, puede ser la detección de rostros en una imagen, la respuesta a un cambio en la base de datos, respuesta a una llamada de API o postear un Tweet. En general, una acción es invocada en respuesta a un evento y produce una salida observable.

1.3.1 Secuencia de acciones

Múltiples acciones pueden ser encadenadas unas con otras, aún a pesar de estar escritas en diferentes lenguajes, creando lo que se llama *secuencia*. Una secuencia puede ser considerada como una sola acción a efectos de la creación y su invocación.

1.3.2 Lo básico para usar funciones:

- Las funciones aceptan y retornan diccionarios. Estos son pares clave-valor, donde la clave es un *string* y el valor es cualquier valor JSON válido. Los diccionarios, de forma canónica, son representados como objetos JSON cuando son utilizados para una acción via API REST o el CLI `wsk`
- La función debe llamarse `main`, o debe ser exportada explícitamente como el punto de entrada. Las mecánicas dependerán del lenguaje elegido, pero en general el entry point puede ser especificado usando el flag `--main` cuando se utiliza el CLI `wsk`

1.4 Disparadores y reglas

¿Qué es un disparador? Es un canal con nombre para clases o tipos de eventos enviados desde fuentes de eventos.

¿Qué es una regla? Se usan para asociar una regla con una acción. Después de que este tipo de asociación es creada, cada vez que un disparador es enviado, la acción asociada es invocada.

¿Qué son las fuentes de eventos? Son servicios que generan eventos que usualmente indican cambios en los datos, o que llevan ellos mismos datos. Algunos ejemplos comunes son:

- Mensajes que llegan en una cola de mensajes
- Cambios en bases de datos
- Cambios en almacenamiento de documentos
- etc

1.5 ¿Cómo funciona OpenWhisk?

En esta sección intentaré dar una breve pero lo más acertada posible descripción de lo que pasa cuando uno crea una acción, y luego la invoca.

Para dar un poco de contexto, vamos a crear una acción simple en nodejs, y luego a invocarla.

Creamos un archivo *action.js* que contiene el siguiente código que va a imprimir "Hola mundo" a stdout, y retorna un JSON conteniendo "Mundo" bajo la clave "hola".

```
function main(){
    console.log('Hola mundo')
    return {hola: 'Mundo'}
}
```

Creamos esta acción con el comando

```
$ wsk action create -i myAction action.js
```

Yla invocamos con

```
$ wsk action invoke -i myAction --result
```

1.5.1 El curso interno de procesamiento

¿Qué pasa en verdad detrás de escena en OpenWhisk?
https://github.com/apache/openwhisk/raw/master/docs/images/OpenWhisk$_flow_of_processing.png$

Entrando al sistema: nginx La API orientada al usuario está completamente basada en HTTP, y sigue un diseño RESTful. Como consecuencia, el comando enviado via el CLI `wsk` es esencialmente un request HTTP que le pega al sistema OpenWhisk.
El primer punto de entrada es **nginx**, "un server proxy HTTP e inverso".

Entrando al sistema: Controller Sin tener mucho para hacer con el request HTTP, nginx lo reenvía hacia el Controller. Es una implementación basada en Scala del REST API, basado en **Akka** y en **Spray**, y que sirve como la interfaz de todo lo que un usuario puede hacer.
El controller primero se encarga de entender lo que el usuario quiere hacer. Lo hace basado en el método HTTP que usamos en el request HTTP. Si se da cuenta que estamos haciendo un POST a una acción existente, el Controller lo traduce como la **invocación de una acción**

Autenticación y Autorización: CouchDB Ahora, el Controller verifica quien envía el request (*Autenticación*) y si tenemos los privilegios necesarios para hacer lo que queremos hacer (*Autorización*). Las credenciales incluídas en el request son verificadas contra los **subjects** en una instancia de CouchDB, en donde se verificará que el usuario exista en la base de datos de OpenWhisk, y que tiene los privilegios necesarios.

Obteniendo la acción: CouchDB, de nuevo Ahora que el Controller está seguro de que el usuario tiene los permisos necesarios, ahora sí carga la acción desde la base de datos **whisks** en CouchDB.

¿Quién está ahí para invocar la acción?: El balanceador de Cargas El balanceador de cargas (Load Balancer), que es parte del Controller, tiene una visión más global de los ejecutores disponibles en el sistema checkeando su estado de salud continuamente. Estos ejecutores son llamados **invocadores** (*invokers*). El Load Balancer, sabiendo qué invokers están disponibles, elige a uno de ellos para que realice la acción pedida.

Por favor, formen una fila: Kafka El controller y el invoker se comunican a través de mensajes que se guardan en buffer y son persistidos por **Kafka**, quien se lleva la carga de guardar el buffer en memoria, para que no sea responsabilidad del Controller y el Invoker. Además, se encargará de que los mensajes no se pierdan en caso de una falla del sistema.
Entonces, para que la acción sea invocada, el Controller publica un mensaje en Kafka, que contiene la acción y los parametros para pasarle a esa acción. Ese mensaje está dirigido al Invoker que el Controller eligió antes.
Una vez que Kafka confirmó que tiene el mensaje, el request HTTP al usuario es respondido con un **ActivationID**. El usuario lo usará luego para obtener acceso a los resultados de esta invocación específica. Notar que este se trata de un modelo asíncrono, donde la request HTTP termina una vez que el sistema aceptó la request para invocar una acción.

Realmente invocando el código de una vez: Invoker El **Invoker** es el corazón de OpenWhisk. Su tarea es invocar una acción. Tamién está implementado en Scala.
La forma de trabajar es usando **Docker**. Básicamente, para cada acción invocada, un container de Docker es creado, el código es inyectado, ejecutado usando los parámetros, se obtiene el resultado y el container es destruido.

Guardando los resultados: CouchDB de nuevo Una vez el resultado es obtenido por el Invoker, es guardado en la base de datos **activations** como una activación bajo el *ActivationId* que se mencionó antes. Esta base de datos vive en CouchDB.

2 Documentación OpenWhisk

Instalación de Apache OpenWhisk

1 Prerrequisitos para utilizar OpenWhisk local

1. Tener una instancia corriendo de kubernetes, en la documentación oficial se recomienda usar Kind. (fuente documentación de OpenWhisk)

2. Para lograr el punto anterior, he instalado kind, tal como lo indica su documentación (Para instalarlo no hubo ningún problema, pero al momento de querer levantar el primer cluster, decidí hacerlo desde mi casa, porque para ello necesita descargar un docker de nodejs, que pesa más de 400 MB)

2 Pasos para la instalación

1. Descargar e instalar kind, siguiendo los pasos de la documentación

```
$ curl -Lo ./kind https://github.com/kubernetes-sigs/
   kind/releases/download/v0.6.0/kind-$(uname)-amd64

$ chmod +x ./kind

$ mv ./kind /some-dir-in-your-PATH/kind #(Yo use /usr/
   bin, pero tuve que ejecutar el comando con sudo)
```

2. Seguir el paso a paso de aquí (es la documentación en GitHub oficial)

    1. Primero, crear el archivo *kind-cluster.yml* con el siguiente contenido:

```
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha3
nodes:
- role: control-plane
- role: worker
  extraPortMappings:
    - hostPort: 31001
      containerPort: 31001
- role: worker
```

    2. Crear el cluster ejecutando:

```
$ kind create cluster --config kind-cluster.yaml
```

1. Instalar `kubectl` siguiendo los pasos de la documentación
2. Configurar `kubectl` (funciona en bash/zsh)

```
$ export KUBECONFIG="$(kind get kubeconfig-path)"
```

3. Ahora falta etiquetar los dos nodos *worker* para que uno quede reservado para el *invoker* y el otro se usará para correr el resto del sistema OpenWhisk

```
$ kubectl label node kind-worker openwhisk-role=
    core

$ kubectl label node kind-worker2 openwhisk-role=
    invoker
```

4. Configurar OpenWhisk.
    1. Obtener el internalIP del worker-node con el comando :

```
$ kubectl describe node kind-worker | grep
    InternalIP: | awk '{print $2}'
```

    2. Crear un archivo *mycluster.yaml*, reemplazando el valor **whisk.ingress.apiHostName** con la salida del comando anterior. Debería quedar así:

```
whisk:
  ingress:
    type: NodePort
    apiHostName: <INTERNAL_IP>
    apiHostPort: 31001
invoker:
  containerFactory:
    impl: "kubernetes"

nginx:
  httpsNodePort: 31001
```

3. Instalar Helm v2.14.3
    1. Descargar esta versión desde la sección releases
    2. Descomprimir el archivo (quedará una carpeta llamada *linux-amd64/*)

3. Mover el archivo *linux-amd64/helm* a alguna carpeta que se encuentre en la variable PATH (yo lo moví a */usr/bin/helm*)
4. Inicializar `Helm Tiller` (o instalarlo) en el cluster de kind:

```
$ helm init
```

5. Fijarse si Helm está corriendo correctamente (vamos a listar los pods, y checkear si ahí está el pod *tiller-deploy*):

```
$ kubectl get pods -n kube-system
```

6. Una vez esté el pod corriendo, debemos darle los permisos necesarios al usuario `Helm`:

```
$ kubectl create clusterrolebinding tiller-cluster
    -admin --clusterrole=cluster-admin --
    serviceaccount=kube-system:default
```

4. Ahora, toca desplegar OpenWhisk (¡Finalmente!):

```
$ helm install ./helm/openwhisk --namespace=openwhisk
   --name=owdev -f mycluster.yaml
```

5. Instalar el CLI de OpenWhisk, llamado *wsk* de la siguiente forma:
    1. Descargarse el binario comprimido de esta página
    2. Descomprimirlo y mover el binario *wsk* a alguna carpeta que se encuentre en PATH. (Yo lo moví a */home/uriel/.local/bin/wsk*)
6. Configurar el CLI instalado en el paso anterior, reemplazando en el comando las palabras `whisk.ingress.apiHostName` y `whisk.ingress.apiHostPort` por los valores que están en el archivo *mycluster.yaml*

```
$ wsk property set --apihost <whisk.ingress.
   apiHostName>:<whisk.ingress.apiHostPort>

$ wsk property set --auth 23bc46b1-71f6-4ed5-8c54-816
   aa4f8c502:123
   zO3xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyGVCGuMDGIwP
```

7. Para hacer una prueba, podemos crear una acción en Python:

    1. Primero creamos un archivo *hello.py* con el siguiente código dentro:

```
def main(args):
    name = args.get("name", "stranger")
    greeting = "Hello " + name + "!"
    print(greeting)
    return {"greeting": greeting}
```

    2. Creamos una acción con el comando:

```
$ wsk action create -i helloPython hello.py
```

    El flag `-i` (`--insecure`) le está diciendo a OpenWhisk que no chequee los certificados. Esto lo hacemos así con confianza porque se trata de un ambiente local de desarrollo.

    3. Ahora, hacemos una llamada a la acción:

```
$ wsk action invoke -i helloPython --param name
   World
```

Para conocer más sobre cómo crear e invocar acciones, aquí está la documentación oficial.

3 Problemas que fueron surgiendo:

1. Cuando quise levantar el cluster de K8s por primera vez, falló porque intentó pegarle al puerto 8080 que estaba rechazando las conexiones. He tenido que revisar uno cuál configuración faltó realizar.

    - Solución: Al momento de configurar kubectl, hay que usar el comando `export KUBECONFIG="$(kind get kubeconfig-path)"` en lugar del comando sin el export.

2. Cuando quise crear una nueva acción en OpenWhisk, tiraba errores de que los certificados estaban vencidos:

```
   error: Unable to invoke action 'helloPython':
      Post https://172.17.0.2:31001/api/v1/
      namespaces/_/actions/helloPython?blocking=
      true&result=true: x509: certificate has
      expired or is not yet valid.
```

- Solución: mandarle el flag `-i` (`--insecure`), que saltea la verificación de certificados. He decidido hacerlo así porque se trata nada más que de una instalación local, para pruebas. (Aquí dejo la documentación oficial en donde se menciona esta solución)

4 Tecnologías utilizadas:

- KinD: Kubernetes in Docker, es una herramienta para correr clústeres de Kubernetes locales, contenedores de Docker.
- Helm: Es una herramienta para gestionar Charts. Charts son paquetes de recursos de Kubernetes pre-configurados. Es similar a los gestores de paquetes como `aptitude`, `pacman`, `yum`, etc, pero para Kubernetes.
- kubectl: Es la herramienta de línea de comandos de Kubernetes, que permite desplegar y gestionar aplicaciones en Kubernetes. Esta herramienta permite inspeccionar recursos del clúster; crear, eliminar, y actualizar componentes; explorar un nuevo clúster; y arrancar aplicaciones de ejemplo.
- wks cli: Es la interfaz de línea de comandos de OpenWhisk, que sirve para interactuar con los servicios de OpenWhisk.

5 Bibliografía para documentación de herramientas de instalación:

- Documentación oficial KinD
- Documentación oficial Helm
- Documentación para instalar kubectl
- Documentación para instalar wsk cli
- Documentación de OpenWhisk para instalar en Kubernetes
- Documentación de OpenWhisk para instalar en kind y tener una instancia local