Chapter 1. Serverless and OpenWhisk Architecture

Welcome to the world of Apache OpenWhisk, an open source serverless platform designed to make it simple to develop applications in the cloud. The project was developed in the open by the Apache Software Foundation, so the correct name is “Apache OpenWhisk,” but for simplicity we’ll use “OpenWhisk” throughout.

Note that “serverless” does not mean “without a server”—it means “without managing the server.” Indeed, we will learn how to build complex applications without being concerned with installing and configuring the servers to run the code; we only have to deal with the servers when we first deploy the platform.

A serverless environment is most suitable for applications needing processing “in the cloud” because it allows you to split your application into multiple simpler services. This approach is often referred to as a “microservices” architecture.

To begin with, we will take a look at the architecture of OpenWhisk to understand its strengths and weaknesses. After that we’ll discuss the architecture itself, focusing on the serverless model to show you what it can and cannot do.

We’ll wrap up this chapter by comparing OpenWhisk with another widely used similar architecture, Java EE.

The problems previously solved by Java EE application servers can now be solved by serverless environments, only at a greater scale (even hundreds of servers) and with more flexibility (not just with Java, but with many other programming languages).

TIP

Since the project is active, new features are added almost daily. Be sure to check the book's website for important updates and corrections.

OpenWhisk Architecture

Apache OpenWhisk, as shown in Figure 1-1, is a serverless open source cloud platform. It works by executing functions (called actions) in response to events. Events can originate from multiple sources, including timers, databases, message queues, or websites like Slack or GitHub.

OpenWhisk accepts source code as input that provisions executing a single command with a command-line interface (CLI), and then delivers services through the web to multiple consumers, such as other websites, mobile applications, or services based on REST APIs.

Figure 1-1. How Apache OpenWhisk works

Functions and Events

OpenWhisk completes its tasks using *functions*. A function is typically a piece of code that receives some input and provides an output in response. It is important to note that a function is generally expected to be *stateless*.

Backend web applications are *stateful*. Just think of a shopping cart application for e-commerce: while you navigate the website, you add your items to the basket to buy them at the end. You keep a state, which is the contents of the cart.

But being stateful is expensive; it limits scalability because you need a place to store your data. Most importantly, you will need something to synchronize the state between invocations. When your load increases, this “state-keeping” infrastructure will limit your ability to grow. If you are stateless, you can usually add more servers because you do not have the housekeeping of keeping the state in sync among the servers, which is complex, expensive, and has limits.

In OpenWhisk, and in serverless environments in general, the functions must be stateless. In a serverless environment you can keep state, but not at the level of a single function. You have to use some special storage that is designed for high scalability. As we will see later, you can use a NoSQL database for this.

The OpenWhisk environment manages the infrastructure, waiting for something important to occur.

This something important is called an *event*. Only when an event happens a function is invoked.

Event processing is actually the most important operation the serverless environment manages. We will discuss in detail next how this happens. Developers want to write code that responds correctly when something happens—e.g., a request from the user or the arrival of new data—and processes the event quickly. The rest belongs to the cloud environment.

In conclusion, serverless environments allow you to build your application out of simple stateless functions, or *actions* as they are called in the context of OpenWhisk, that are triggered by events. We will see later in this chapter what other constraints those actions must satisfy.

Architecture Overview

Now that we know what OpenWhisk is and what it does, let's take a look at how it works under the hood. Figure 1-2 provides a high-level overview.

Figure 1-2. An example deployment with actions in multiple languages

In Figure 1-2, the big container in the center is OpenWhisk itself. It acts as a container of actions. We will learn more about the container and these actions shortly, but as you can see, actions can be developed in many programming languages. Next, we'll discuss the various options available.

NOTE

The “container” schedules the actions, creating and destroying them as needed needed, and it will also scale them, creating duplicates in response to an increase in load.

Programming Languages for OpenWhisk

You can write actions in many programming languages. Typically, *interpreted* programming languages are used, such as JavaScript (actually, Node.js), Python, or PHP. These programming languages give immediate feedback because you can execute them without a compilation step. While these are higher-level languages and are easier to use, they are also slower than compiled languages. Since OpenWhisk is a highly responsive system (you can immediately run your code in the cloud), most developers prefer to use those interpreted languages as their use is more interactive.

TIP

While JavaScript is the most widely used language for OpenWhisk, other languages can also be used without issue.

In addition to purely interpreted (or more correctly, compiled-on-the-fly) languages, you can also use the *precompiled interpreted languages* in the Java family such as Java, Scala, and Kotlin. These languages run on the Java Virtual Machine (JVM) and are

distributed in an intermediate form. This means you have to create a *.jar* file to run your action. This file includes the so-called “bytecode” OpenWhisk executes when it is deployed. A JVM actually executes the action.

Finally, in OpenWhisk you can use compiled languages. These languages use a binary executable that runs on “bare metal” without interpreters or virtual machines (VMs). These binary languages include Swift, Go, and the classic C/C++. Currently, OpenWhisk supports Go and Swift out of the box. However, you can use any other compiled programming language as long as you can compile the code in Linux *elf* format for the *amd64* processor architecture. In fact, you can use any language or system that you can package as a Docker image and publish on Docker Hub: OpenWhisk is able to retrieve this type of image and run it, as long as you follow its conventions.

NOTE

Each release of OpenWhisk includes a set of runtimes for specific versions of programming languages. For the released combinations of programming languages and versions, you can deploy actions using the switch `--kind` on the command line (e.g., `--kind nodejs:6` or `--kind go:1.11`). For single file actions, OpenWhisk will select a default runtime to use based on the extension of the file. You can find more runtimes for programming languages or versions not yet released on Docker Hub that can be used with the switch `--docker` followed by the image name.

Actions and Action Composition

OpenWhisk applications are collections of actions. Figure 1-3 shows how they are assembled to build applications.

Figure 1-3. Overview of OpenWhisk action runtimes

An action is a piece of code, written in one of the supported programming languages (or even an unsupported language, as long as you can produce an executable and package it in a Docker image), that you can invoke. On invocation, the action will receive some information as input.

To standardize parameter passing among multiple programming languages, OpenWhisk uses the widely supported JavaScript Object Notation (JSON) format, because it's pretty simple and there are libraries to encode and decode this format available for basically every programming language.

The parameters are passed to actions as JSON objects serialized as strings that the action receives when it starts and is expected to process. At the end of the processing, each action must produce a result, which is returned as a JSON object value.

You can group actions in *packages*. A package is a unit of distribution. You can share a package with others using *bindings*. You can also customize a package, providing parameters that are different for each binding.

Action Chaining

Actions can be combined in many ways. The simplest way is chaining them into *sequences*.

Chained actions use as input the output of the preceding actions. Of course, the first action of a sequence will receive the parameters (in JSON format), and the last action of the sequence will produce the final result as a JSON string. However, since not all the flows can be implemented as a linear pipeline of input and output, there is also a way to split the flows of an action into multiple directions. This feature is implemented using triggers and rules. A *trigger* is merely a named invocation. By itself a trigger does nothing. However, you can associate the trigger with one or more actions using *rules*. Once you have created the trigger and associated some action with it, you can *fire* the trigger by providing parameters.

NOTE

Triggers cannot be part of a package. but they can be part of a namespace, as we'll see in Chapter 3.

The actions used to fire a trigger are called a *feed* and must follow an implementation pattern. In particular, as we will learn in "Observer", actions must implement an Observer pattern and be able to activate a trigger when an event happens.

When you create an action that follows the Observer pattern (which can be implemented in many different ways), you can mark the action as a feed in a package. You can then combine a trigger and a feed when you deploy the application, to use a feed as a source of events for a trigger (and in turn activate other actions).

How OpenWhisk Works

Now that you know the different components of OpenWhisk, let's look at how OpenWhisk executes an action.

The process is straightforward for the end user, but internally it executes several steps. We saw before the user visible components of OpenWhisk. We are now going to look under the hood and learn about the internal components. Those components are not visible by the user but the knowledge of how it works is critical to use OpenWhisk correctly. OpenWhisk is "built on the shoulders of giants," and it uses some widely known and well-developed open source projects.

These include:

Nginx

A high-performance web server and reverse proxy

CouchDB

A scalable, document-oriented NoSQL database

Kafka

A distributed, high-performing publish/subscribe messaging system

All the components are Docker containers, a format to package applications in an efficient but constrained, virtual machine–like environment. They can be run any environment supporting this format, like Kubernetes.

Furthermore, OpenWhisk can be split into some components of its own:

Controller

Managing entities, handling trigger fires, and routing actions invocations

Invoker

Launching the containers to execute the actions

Action Containers

Actually executing the actions

In Figure 1-4 you can see how the processing happens. We are going to discuss it in detail, step by step.

Figure 1-4. How OpenWhisk processes an action

NOTE

Basically, all the processing done in OpenWhisk is asynchronous, so we will go into the details of an

asynchronous action invocation. Synchronous execution fires an asynchronous action and then waits for the result.

Nginx

Everything starts when an action is invoked. There are different ways to invoke an action:

- From the web, when the action is exposed as a web action
- When another action invokes it through the API
- When a trigger is activated and there is a rule to invoke the action
- From the CLI

Let’s call the *client* the subject who invokes the action. OpenWhisk is a RESTful system, so every invocation is translated to an HTTPS call and hits the so-called “edge” node. The edge is actually the web server and reverse proxy Nginx. The primary purpose of Nginx is to implement support for the HTTPS secure web protocol, so it deploys all the certificates required for secure processing. Nginx then forwards the requests to the actual internal service component, the controller.