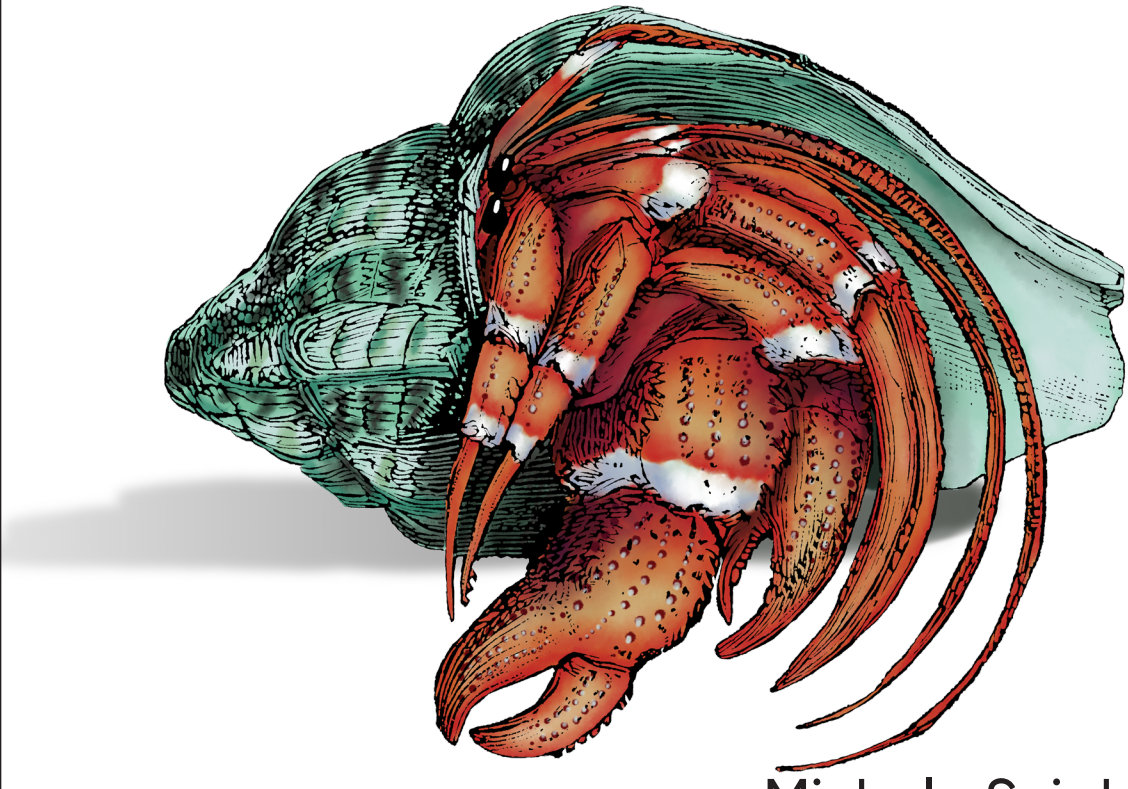
O'REILLY®

Learning Apache OpenWhisk

Developing Open Serverless Solutions



Michele Sciabarrà

O'REILLY®

Learning Apache OpenWhisk

Serverless computing greatly simplifies software development. Your team can focus solely on your application while the cloud-based serverless platform manages the deployment and scaling for you. This practical guide shows you step-by-step how to build and deploy complex applications in a flexible, multicloud, multilanguage serverless environment using Apache OpenWhisk.

You'll learn how this platform enables you to pursue a vendor-independent approach using OpenWhisk running in Kubernetes as your cloud operating system. Michele Sciabarrà demonstrates how to build a serverless application using classical design patterns and the programming languages that best fit your task. You'll start by building a simple serverless application before diving into the more complex aspects of the OpenWhisk platform.

- **Examine OpenWhisk's serverless architecture, including the use of packages, actions, sequences, triggers, rules, and feeds**
- **Interface with OpenWhisk features using the command line or a JavaScript API**
- **Design applications using common Gang of Four design patterns**
- **Learn how to test and debug your code in a serverless environment**
- **Learn how to use JavaScript, Python, and Go for developing your serverless applications**
- **Learn about CouchDb and Kafka integrations with OpenWhisk**
- **Install OpenWhisk in Kubernetes with a compelete step-by-step guide**

"This book is a nice starting point for learning OpenWhisk. It covers all the main features and includes lots of real examples of how to apply these features to your everyday projects."

—Roberto Diaz
Software Developer at The Agile Monkeys and Apache OpenWhisk contributor

Michele Sciabarrà is a veteran of information technology and is currently CEO of *Sciabarra.com*, a consultancy focused on Kubernetes and Serverless solutions. He's also a contributor to the Apache OpenWhisk project, most notably as the author of the high-performance ActionLoop runtime for Go, Swift, Rust, Java, and other programming languages.

CLOUD / SYS ADMIN

US $69.99 CAN $92.99

ISBN: 978-1-492-04616-5

Twitter: @oreillymedia
facebook.com/oreilly

Learning Apache OpenWhisk

Developing Open Source Serverless Solutions

Michele Sciabarrà

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Learning Apache OpenWhisk
by Michele Sciabarrà

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Virginia Wilson and John Devins
Production Editor: Nan Barber
Copyeditor: Christina Edwards
Proofreader: Rachel Head
Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

July 2019: First Edition

Revision History for the First Edition
2019-07-01: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492046165* for release details.

978-1-492-04616-5

[LSI]

Table of Contents

Part II. Advanced OpenWhisk Development

Foreword

Amazon Web Services changed the cloud computing landscape in November of 2014 when it launched a new service called Lambda. It offered developers the intriguing possibility of processing thousands of events concurrently, simply by registering functions as event handlers. With Lambda, functions execute on-demand and scale instantly and costs are proportional to actual resource utilization. This new model of computing was dubbed "serverless" because it let developers eschew all aspects of server-side development and operations—including infrastructure management, resource provisioning, and scaling.

Amazon showed that in the era of managed infrastructure and services, cloud providers can free developers from low-level operational burdens and let them focus on what matters most: delivering real business value to their organizations. Serverless computing is the foundation of a cloud-native transformation that is ongoing in the industry. It is the way cloud applications are being built and will largely be built in the future. There are now more than a trillion functions processed every month on Amazon alone for a wide variety of applications from IoT, web applications, machine learning, and high-performance computing.

At IBM Research, my group took notice of the Lambda announcement, and we quickly realized the value of the serverless promise. So in February 2015, we set out to build a serverless platform for the IBM Cloud. The project was code-named Whisk, as in "to move nimbly and quickly." It was later branded OpenWhisk when we open-sourced the code to GitHub, nearly a year to the day from our first commit. Today, the project is part of the Apache Software Foundation Incubator and it's the premiere open-source alternative to Amazon's Lambda. Apache OpenWhisk is ready for Enterprise workloads and offers a powerful programming model for building entire serverless applications. It powers serverless product offerings from IBM and Adobe, and it's used in private deployments in some of Asia's largest mobile and internet providers. The Apache OpenWhisk community consists of more than 215 contributors from around the world, and is among the top 30 Apache Foundation projects ranked by their GitHub stars. Many of the top OpenWhisk contributors are now also shaping

the future of other emerging serverless platforms such as Google's Knative project, which aims to bring a serverless experience to the increasingly popular Kubernetes platform.

In this book, Michele Sciabarrà delivers a thorough exposition of Apache OpenWhisk tailored to the needs of developers and operators. For developers eager to adopt a serverless methodology, this book explains the emerging computational patterns and current limitations. It also illustrates several examples of applying the technology to solve real-world challenges. For operators who want to deliver a serverless experience for their organizations, Michele carefully reveals the OpenWhisk architecture, and provides a go-to guide for deployment and operations, particularly for a Kubernetes-based deployment.

Michele draws on his own contributions to the Apache OpenWhisk project in writing this book. His work has dramatically improved the performance of serverless functions developed in Go, PHP, Python, Swift, and more. His insightful and practical approach is well-suited to the developer who wants get up to speed quickly in harnessing the power of serverless computing.

For those of us at the frontier of the serverless journey, there is no doubt that we are witnessing the dawn of a new Cloud Computer, with functions at the cornerstone of a new instruction set architecture. This model of computing is powerful and disruptive, and yet much work remains to be done. History has shown that as new computer abstractions emerge, innovation follows.

— Rodric Rabbah
CTO and Cofounder Nimbella Corporation
April, 2019

Preface

This book is for developers who want to learn to use Apache OpenWhisk, a mature, multilanguage, serverless development platform. It provides the knowledge needed to build complex, well-structured, polyglot serverless applications that can be deployed in any cloud or even on-premises.

Why Serverless?

For a long time, companies built their own data centers and bought hardware to install their web applications, mobile backends, or data processing pipelines. But the high costs of building and maintaining data centers eventually led to people renting servers from third parties to reduce costs. Servers are frequently partitioned to "virtual machines," allowing businesses to pay only for what they need (as opposed to renting a whole server). Eventually, this concept evolved into what today we generically call "the cloud." The cloud at its core is mostly a "server on demand" service. Modern cloud providers offer a wide range of services but have also become increasingly complex. In particular, one source of this complexity is the need to provision and manage servers. Servers are cattle not easy to raise. They require continuous care and control and must be monitored, cleaned, updated, and occasionally destroyed and rebuilt. Like cattle, they also tend to grow and multiply quickly.

Developers want to forget the server ("serverless") and develop their application as native citizen of the cloud ("cloud-native"). The focus here is to push the burden of managing the servers onto the shoulders of a platform deployed in the cloud. Most software developers want to upload their code and immediately have it up and running.

As a result, all the major cloud providers now offer some form of Function as a Service that hides the servers and only needs the code to run—in short, a way to use their cloud serverless.

Why Apache OpenWhisk?

Serverless computing was pioneered by Amazon Web Services (AWS), with its Lambda service. Amazon Lambda is, as you might expect, highly tailored to the AWS offering. However, serverless computing is not (and cannot be) limited to one cloud provider. As a natural consequence, some open source serverless projects have emerged. Today, there are many available platforms for serverless computing, which can run in multiple clouds. The focus of this book is one of those: Apache OpenWhisk.

Apache OpenWhisk was originally developed by IBM but its code base was later donated to the Apache Software Foundation and released under a commercially friendly open source license, the Apache License 2.0. This makes it "seriously" open source, free of commercial limitations, friendly to commercial ventures adopting it, and maintained in the long run.

Because any Apache project has to have a working code base as well as an active community, adopters of Apache software can trust they will not be alone in using it. Also, Apache-supported software has a detailed list of compliance rules allowing everyone to use the released software without risking getting caught in the "noncommercial" traps that some other open source licenses have.

In addition to those advantages of being an Apache project, OpenWhisk is used in production and powers the cloud functions of the IBM Cloud as well as Adobe's I/O runtime cloud services. Apache OpenWhisk also supports many programming languages, including Node.js, Python, Java, Go, Swift, PHP, and Ruby. More languages are also in the works. In short, it is a stable and sophisticated production-ready serverless platform.

What You Will Learn

This book is for developers, so you need to be familiar with coding and programming languages. The book is split into two parts—the first part is introductory and the other more advanced.

In the first part, I assume you only know the JavaScript programming language and the basics of web development, like HTML. I do not assume you have experience in serverless development.

In this section you will learn how to create OpenWhisk applications from scratch. After exploring the OpenWhisk architecture, we will create a simple contact form for a static website.

Then we'll explore the CLI and API of OpenWhisk and rebuild the same application in a highly engineered way, splitting it into cooperating actions and applying a bunch

of design patterns. You will learn about the building blocks you need to develop your applications along with examples and best practices.

In this book we emphasize the importance of testing, so the first part ends with a chapter entirely devoted to unit testing, mocking, and snapshot testing.

The second part of the book is more advanced, and here I assume you know more programming languages. Two chapters in the second part use Python for coding examples, and another two chapters use Go. We'll discuss the peculiarities of developing OpenWhisk applications in those programming languages.

In the second part we'll also explore integration with essential external services such as databases (CouchDB) and messaging queues (Kafka). Last but not least, the final chapter covers installing OpenWhisk in Kubernetes, including the installation of Kubernetes itself. This last chapter is more oriented to system administrators but could also be helpful for those unfamiliar with Linux because it provides step-by-step installation instructions.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://learning-apache-openwhisk.github.io*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning Apache OpenWhisk* by Michele Sciabarrà (O'Reilly). Copyright 2019 Michele Sciabarrà, 978-1-492-04616-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text

and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/learn-apache-openwhisk*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I want to say thank you to Rodric Rabbah and Carlos Santana for their support and mentoring. If it were not for them, all I would have ended up with is the single line of code I submitted to fix the Vagrant build on Windows, my first contribution to the OpenWhisk project. Instead, thanks to their support and encouragement, I built a new action proxy that now powers multiple OpenWhisk runtimes, and I ended up writing a book on Apache OpenWhisk.

I also want to thank the reviewers. Carlos and Rodric, of course, were the first to review my early chapters, but Rob Allen, Dragos Dascalita, and Vincent Hou, also provided invaluable suggestions to improve my writing and correct the errors they saw.

Last but not least, I want to thank some members of the Apache community involved with the OpenWhisk project: Dave Grove, Justin Halsall, James Thomas, Markus Thömmes, Matt Rutkowski, Priti Desai, and all the others I've forgotten to name here.

PART I

Introducing OpenWhisk Development

In this first part, you are going to learn how to develop OpenWhisk applications. We'll start with the architecture of the system and the CLI. Then you will use the JavaScript API, debugging techniques, and design patterns developed specifically in JavaScript.

JavaScript is undoubtedly one of the most important and widely used programming languages available. It is also the most common language used by frontend developers, those who may need to add backend logic and don't want to deal with the complexities of managing servers. Most people that use OpenWhisk usually start with JavaScript to keep things simple.

To hit the ground running, we'll start by developing a simple form application. Once you know the basics, we will rewrite the application using design patterns. We'll also cover unit testing and "mocking," used to test software that runs in the cloud.

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).

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.

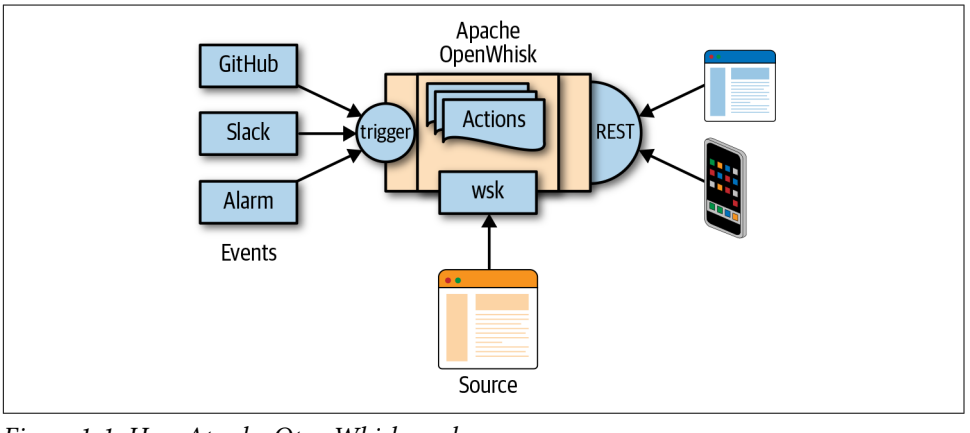


Figure 1-1. How Apache OpenWhisk works

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 sin-

gle 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.

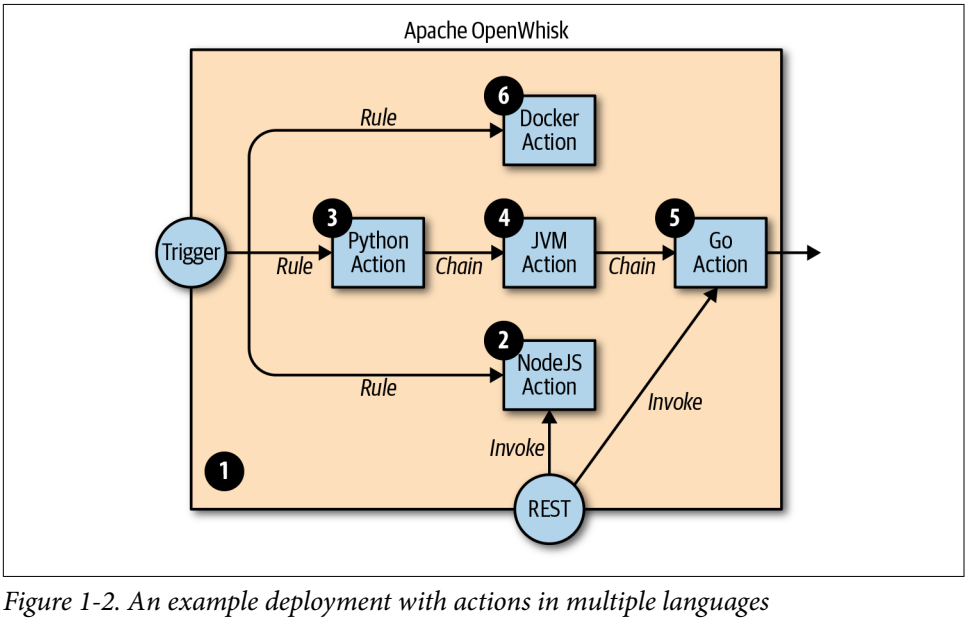


Figure 1-2. An example deployment with actions in multiple languages

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.

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.

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.

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.

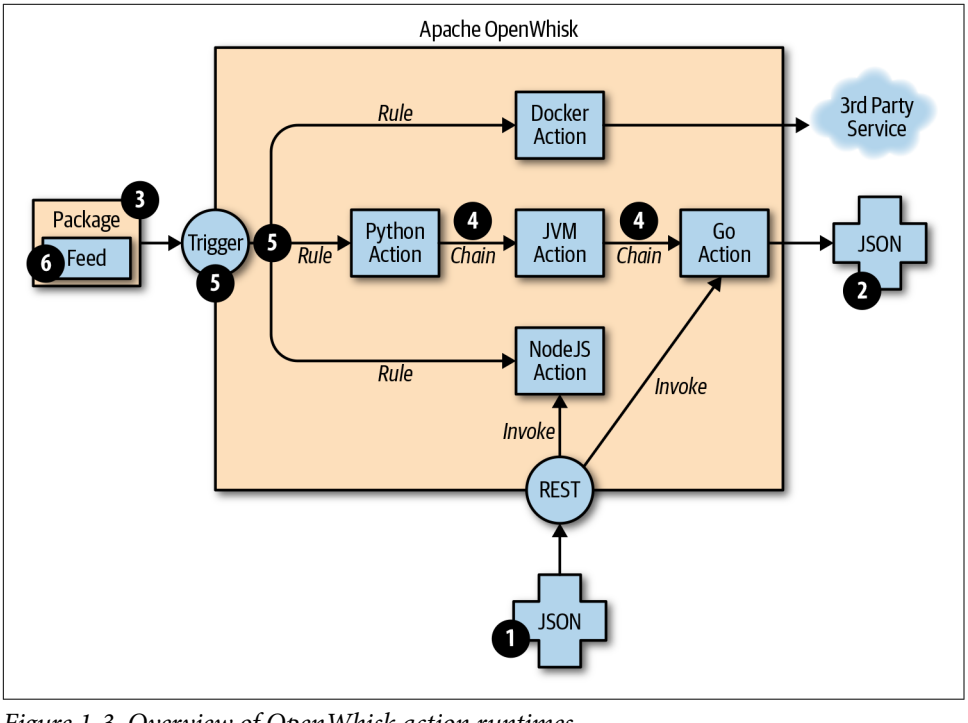


Figure 1-3. Overview of OpenWhisk action runtimes

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.

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" on page 110, 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.

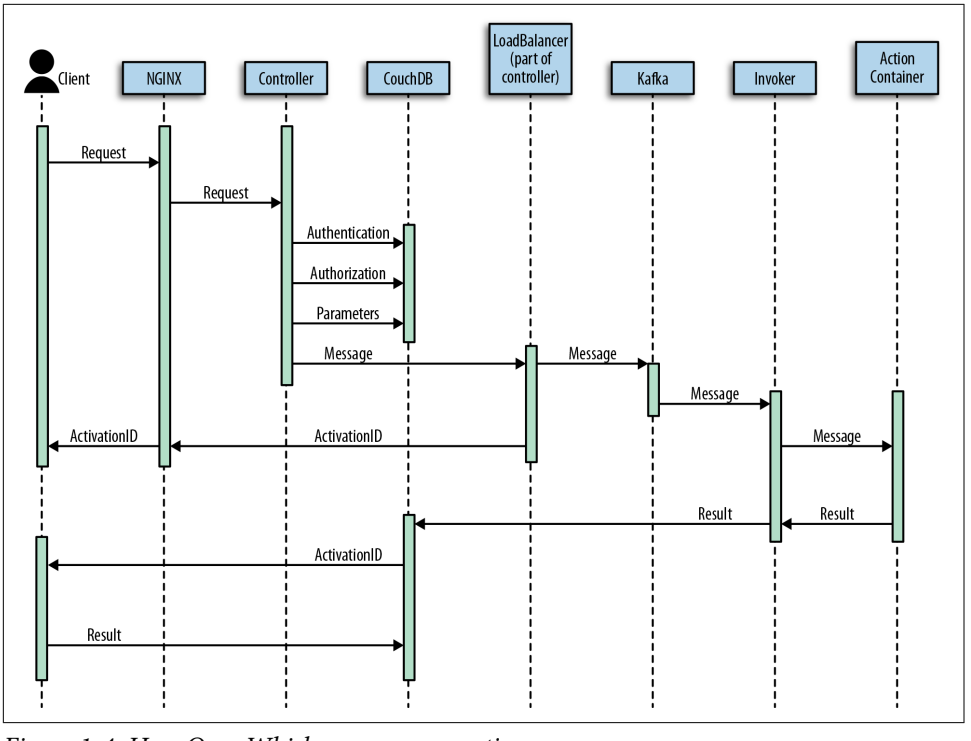


Figure 1-4. How OpenWhisk processes an action

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 pur-

pose 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.

Controller

Before executing the action, the controller checks whether it can execute the action and initialize it correctly:

1. It needs to be sure it can execute the action, so it must *authenticate* the request.
2. Once the origin of the request has been identified, it needs to be *authorized*, verifying that the subject has the appropriate permissions.
3. The request must be enriched with some additional parameters that, as we will see, are provided as part of action configuration.

To perform all those steps the controller consults the database, which in OpenWhisk is CouchDB. Once validated and enriched, the action is now ready to be executed, so it is sent to the next component of the processing, the load balancer.

Load Balancer

The job of the load balancer, as its name implies, is to balance the load among the various executors in the system, which are called *invokers* in OpenWhisk.

We already saw that OpenWhisk executes actions in runtimes. The load balancer keeps an eye on the available instances of the action runtime, reuses the existing ones if they are available, or creates new ones if they are needed.

We've arrived at the point where the system is ready to invoke the action. However, you cannot just send your action invocation to an invoker, because it may be busy serving another action. There is also the possibility that an invoker has crashed, or even that the whole system has crashed and is restarting.

So, because we are working in a massively parallel environment that is expected to scale, we have to consider the possibility that we will not have the resources we need to execute the action immediately. In cases like this, we have to *buffer* invocations. OpenWhisk uses Kafka to perform this action. Kafka is a high-performing "publish and subscribe" messaging system that can store your requests until they are ready to be executed. The request is turned into a message addressed to the invoker the load balancer chose for the execution. An action invocation is actually turned in an HTTPS request to Nginx; then it internally becomes a message to Kafka.

Each message sent to an invoker has an identifier called the *activation ID*. Once the message has been queued in Kafka, there are two possibilities: a nonblocking and a blocking invocation.

For a nonblocking invocation, the activation id is sent back as the final answer to the request to the client, and the request completes. In this case, the client is expected to come back later to check the result of the invocation.

For a blocking invocation, the connection stays open: the controller waits for the result from the action and sends the result to the client.

Invoker

In OpenWhisk the invoker is in charge of executing the actions. Actions are actually executed by the invoker in isolated environments provided by Docker containers. As already mentioned, Docker containers are execution environments that resemble an entire operating system, providing everything needed to run an application.

So, from the actions perspective, the environment provided by a Docker container looks like an entire computer (just like a VM). However, execution within containers is much more efficient than in VMs, so they are preferred.

It would be safe to say that, without containers, serverless environments like OpenWhisk would not be possible.

Docker actually uses *images* to create the containers that execute actions. A runtime is really a Docker image. The invoker launches a new image for the chosen runtime and then initializes it with the code of the action. OpenWhisk provides a set of Docker images including support for various languages. The action runtimes also include the initialization logic. They support JavaScript, Python, Go, Java, and similar languages.

Once the runtime is up and running, the invoker passes the action requests that have been constructed in the processing so far. The invoker also manages and stores the logs needed to facilitate debugging.

After OpenWhisk completes the processing, it must store the result somewhere. This place is again CouchDB (where configuration data is also stored). Each result of the execution of an action is then associated with the activation ID, the one that was sent back to the client. Thus, the client can retrieve the result of its request by querying the database with the ID.

Client

The processing described so far is *asynchronous*. This means the client will start a request and forget about it, although it doesn't leave it behind entirely, because it returns an activation ID as the result of an invocation. As we have seen already, the activation ID is used to store the result in the database after the processing. To

retrieve the final result, the client will have to perform a request again later, passing the activation ID as a parameter. Once the action completes, the result, the logs, and other information will be available in the database and can be retrieved.

Synchronous processing is also available. It works the same way as asynchronous processing, except the client will block waiting for the action to complete and retrieve the result immediately.

Serverless Execution Constraints

Serverless applications come with a few constraints and limitations. We call those constraints the *execution model*. You can think of your application as a set of actions, collaborating with each other to meet the purpose of the application. Each action running in a serverless environment will be executed within certain limits, and those limits must be considered when designing the application.

Figure 1-5 shows the most important action execution constraints. All constraints have some value in terms of time or space, (timeout, frequency, memory, disk size, etc.). Some are configurable; others are hardcoded. Typical constraints (which can be different depending on the particular installation you are using) are:

- *Execution time*: max 1 minute per action
- *Memory size*: max 256 MB per action
- *Log size*: max 10 MB per action
- *Code size*: max 48 MB per action
- *Parameters*: max 1 MB per action
- *Result*: max 1 MB per action

Note that execution time and memory size can be configured by the developer of the action using annotations. Other constraints cannot be configured by the user, but they can be configured by the system administrator of OpenWhisk.

Furthermore, there are global constraints:

- *Concurrency*: max 100 concurrent activations can be queued at the same time (configurable)
- *Frequency*: max 120 activations per minute can be requested (configurable)

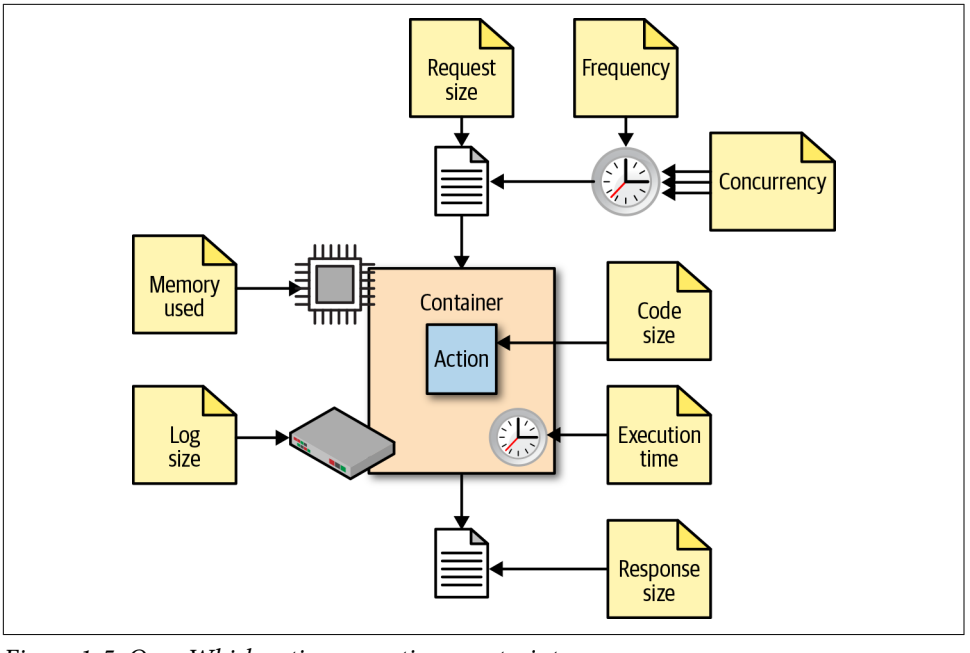


Figure 1-5. OpenWhisk action execution constraints

Global constraints are actually per namespace. You can think of a namespace as a collection of OpenWhisk resources available under a URL prefix that can be accessed using the same API token (so they can invoke each other). In a sense, a namespace is the serverless equivalent of an application, split into multiple related entities.

Let's discuss the qualitative constraints that impact the way you have to develop actions.

Actions Are Functional

As already mentioned, each action must be a function invoked with a single input and must produce a single output. The input is a string in JSON format. The action usually deserializes the string in a data structure specific to the programming language used for the implementation. The runtime generally performs the deserialization, so the developer receives an already-parsed data structure. If you use dynamic languages like JavaScript or Python, you will usually receive something like a JavaScript object or a Python dictionary, which can be efficiently processed using the programming language.

If you use a statically typed language like Java or Go, you may need to put more effort into decoding the input. Libraries for performing such decoding are readily available.

However, some decoding work may be necessary to map the generally untyped arguments to the typed data structure of the language.

The same holds true for returning the output. It must be a single data structure appropriate for the programming language you are using, but it must also be serialized back into JSON format before being returned. Runtimes usually take care of serializing data structures back into strings.

Actions Are Event-Driven

Everything in the serverless environment is activated by events. Your code should execute quickly, do what is requested, and terminate. It is the system that will invoke your code when it is needed. An example of an event is when a user browses the web and invokes the URL of a web action deployed in OpenWhisk. This event triggers an action invocation.

But this is only one possible event. Another example is a request from another action that arrives in a message queue.

Database management is also event-driven. You can perform a query on a database, then wait until an event is triggered when the data arrives.

Websites can originate events, too. For example, you may receive an event:

- When someone pushes a commit on GitHub
- When a user interacts with Slack and sends a message
- When a scheduled alarm is triggered

Actions Do Not Have Local State

Actions are executed in Docker containers. A container is created to execute a single action; it serves a number of requests, and then it is destroyed. As a consequence (by Docker design), the filesystem is ephemeral. Once a container terminates, all the data stored on the filesystem will be removed too. But this does not mean you cannot store anything in files when using a function. You can use files for temporary data storage while executing an application.

Indeed, a container can be used multiple times. For example, if your application needs some data downloaded from the web, an action can can perform the downloading and then save it and make it available to other actions executed in the same container.

What you cannot assume is that the data will stay in the container forever. At some point in time, either the container will be destroyed or another container will be cre-

ated to execute the action. In a new container, the data you downloaded in a previous execution of the action will no longer be available.

In short, you can use local storage as a cache for speeding up further actions, but you cannot rely on the fact that the data you store in a file will persist forever. For long-term persistence, you need to use other means: typically a database or another form of cloud storage.

Actions Are Time-Bound

Actions must complete in the shortest time possible. As already mentioned, the execution environment imposes time limits on the execution time of an action. If the action does not terminate within that timeframe, it will be aborted. This is also true for background actions or threads you may have started.

So, ensure that your code will not keep going for an unlimited amount of time (e.g., when it gets larger input). Instead, you may want to split the processing into smaller chunks and ensure the execution time of your action will stay within the necessary limits.

Also remember that billing can be time-dependent. If your cloud provider supports OpenWhisk with a pay-per-use model, you will be charged for the time your actions take. Faster actions will result in lower costs. When you have millions of actions executing, a few milliseconds can make a difference.

If you install OpenWhisk on your own servers, you are usually only billed for the VMs running them.

Actions Are Not Ordered

Note also that actions are not ordered. If you invoke action A at time X and action B at time Y, with X < Y, action B may be executed *before* A. As a consequence, if the actions have side effects—for example, writing to the database—the side effect of action A may be applied later than that of action B. Furthermore, there is no guarantee an action will be executed entirely before or after another action. They may overlap in time. For example, if you are writing to a database you must be aware that another action may start writing to it too before you are finished. So, you have to provide transaction support.

From Java EE to Serverless

While the serverless architecture may look brand new, it actually has quite a bit in common with existing architectures. In a sense, it is an evolution of those historical architectures.

To better understand the genesis and the advantages of the serverless architecture, it makes sense to compare the OpenWhisk architecture with one of its historical precedents: Java Enterprise Edition, or Java EE.

Java EE itself was a specification. Some of the most prominent implementations of this specification still in broad use today are Oracle WebLogic and IBM WebSphere.

Classic Java EE Architecture

The core idea behind Java EE was to allow the development of large application out of small, manageable parts. It was a technology designed to help application development for large organizations, using the once new and revolutionary Java programming language—hence, the name *Java Enterprise Edition*.

When the Java EE was created, everything was based on the Java programming language, leveraging the VM. At the time Java was considered to be a programming language suitable for building large, scalable applications (meant to replace C++). Scripting languages like Python were not yet largely used, and JavaScript was still in its infancy.

To facilitate the development of these vast and complex applications, Java EE provided a productive (and complicated) infrastructure of services offering many different types of components, each one deployable separately, with many ways for the various parts to communicate with each other. Those services were put together and made available through the use of software packages called *application servers*. Figure 1-6 gives an overview of the JavaEE architecture and its different tiers.

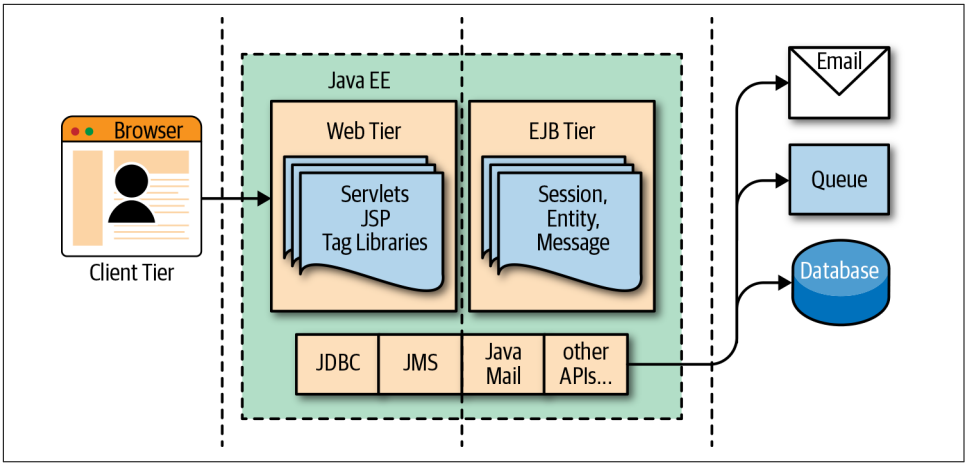


Figure 1-6. Java EE architecture

The client tier was expected to implement the application logic at the client level. Historically Java was used to implement *applets*, small Java components that were downloaded and run in browsers. JavaScript, CSS, and HTML replaced those web components.

In the web tier, the most crucial components were the *servlets*, further specialized in into JavaServer Pages (JSP), tag libraries, etc. Those components defined the web user interface at the server level.

The so-called *business logic*, managing data and connection to other enterprise systems, was expected to be implemented in the business or Enterprise Java Beans (EJB). There were many flavors of EJB, like Entity Beans, Session Beans, Message Beans, etc.

Each component in Java EE was a set of Java classes that implemented an API. The developer wrote those classes and then delivered them to the application server, which loaded the components and ran them in their containers.

The application servers also provided a set of *connectors* to interface the application with the external world, in the enterprise information system (EIS) tier. There were Java connectors allowing applications to interface with virtually any resource of interest, including:

- Databases
- Message queues
- Email and other communication systems

In the Java EE world, application servers provided the implementation of the entire Java EE specification, including APIs and connectors, acting as a one-stop solution for all the development needs of enterprises.

Serverless Equivalent of Java EE

For many reasons, OpenWhisk can be seen as an evolution of Java EE. Both started from the same basic idea: split your application into many small, manageable parts, and provide a system to quickly put together all those pieces.

However, the technological landscape driving the development of serverless environments is different. In today's world:

- Applications are spread among multiple servers in the cloud, requiring virtually infinite scalability.
- We have numerous programming languages, including scripting languages, that are used extensively.
- VM and container technologies are available to wrap and control the execution of programs.
- HTTP can be considered a standard transport protocol.
- JSON is simple and widely used as a universal exchange format.

Figure 1-7 shows the OpenWhisk architecture in a way that is easy to compare with Java EE. It is intentionally similar to Figure 1-6, to make it easier to see the similarities and differences between the two architectures.

Tiers

As you can see, OpenWhisk also has a web tier and a business tier, in addition to a client and an integration tier. While there is no formal separation between the two tiers, in practice OpenWhisk has actions that are directly exposed to the web (web actions) and actions that are not.

Web actions can be considered to belong to the web tier. Other actions, meant to serve events either coming from web actions or triggered by other services in the infrastructure, can be considered business actions, defining a business tier.

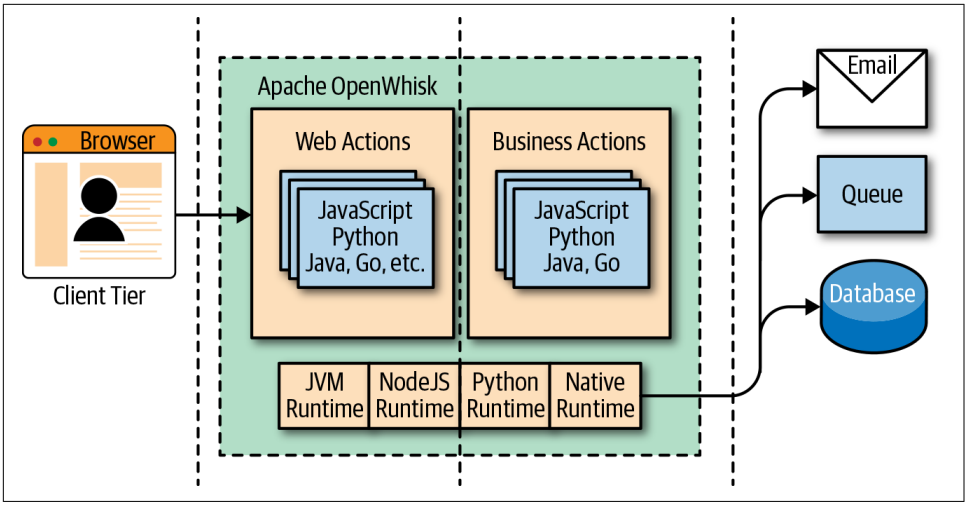


Figure 1-7. OpenWhisk architecture

Components

In Java EE, everything runs in the Java Virtual Machine, and everything must be coded in Java (or at least a programming language that can generate JVM-compatible bytecode). In OpenWhisk, you can code applications in multiple programming languages. We use their runtimes as equivalents of the JVM. Furthermore, those runtimes are wrapped in Docker containers to provide isolation and control over resource usage. This means you can write your components in any (supported) language you like. You are no longer confined to writing your application solely for the JVM. However, a JVM runtime is available, so you can still use Java and its family of languages if you like.

You can now write your code for the JavaScript Virtual Machine, more commonly referred to as *Node.js*.

Under the hood, *Node.js* is an adaptation of the V8 JavaScript interpreter that powers the Chrome browser. It is a fast executor of the JavaScript language, which also has advanced functions like compiling on the fly to native code to speed up execution.

Instead of using VMs, you can also use language-producing native executables like Swift or Go. Go is becoming a popular choice in the serverless world. As long as you compile your application for Linux and AMD64, you can deploy it in OpenWhisk.

APIs

In JavaEE, you have APIs available to interact with the rest of the world, written in Java itself. Basically, in the Java EE model, every interesting resource for writing applications has been adapted to be used by Java.

In OpenWhisk, you have only one API for all the supported programming languages. This is the OpenWhisk API, and it is a RESTful API. You can invoke it over HTTPS using JSON as an interchange format.

This API can even be invoked directly using any HTTP library that can read and write JSON objects as strings. The OpenWhisk API acts as glue for the various components of the platform. All the communications among the different parts in OpenWhisk are performed in JSON over HTTP.

Connectors

In JavaEE, you have connectors for each external system you want to communicate with. For example, if you're going to interact with an Oracle database, you need an Oracle JDBC connector; to communicate with IBM DB2, you need a DB2 JDBC driver, etc. The same holds true for messaging queues, email, and so on.

In OpenWhisk, interactions with other systems are wrapped in *packages*, collections of actions explicitly written to interact with a particular system. You can use any programming language and available APIs and drivers to communicate with packages. For example, if you have a Java driver for a database, you can write a package to interact with it. Packages act as connectors.

In the IBM Cloud, there are packages available to communicate with essential services such as:

- The cloud database Cloudant
- The Kafka messaging system
- The enterprise chat Slack
- Many others, some specific to IBM services

You use the feed mechanism provided by OpenWhisk to hook into those systems.

Application servers

In Java EE, everything is managed by application servers. They are the containers where enterprise applications are meant to be deployed.

In a sense, OpenWhisk itself takes on this role by providing a cloud-based, multi-node, language-agnostic execution service. Using a serverless engine like Open-

Whisk, the cloud becomes a transparent entity where you deploy your code. The environment manages the distribution of applications in the cloud.

The problem then becomes not to install your code, but to install OpenWhisk. Each component of OpenWhisk must be appropriately deployed according to the available resources of the cloud. The installation of OpenWhisk in the cloud is a complex subject, and we will devote Chapter 12 to it.

We know that OpenWhisk runs in Docker containers. However, scaling Docker in a cloud requires you to manage those Docker containers under supervisory systems called *orchestrators*.

There are many orchestrators, but at the moment OpenWhisk supports the following:

- Kubernetes, a widely used orchestration system initially developed by Google
- DC/OS, a cloud operating system built on top of Apache Mesos supporting the management of distributed applications and Docker containers.

Summary

Apache OpenWhisk is a serverless application platform. In this chapter, we learned how the serverless approach to software development makes it easy to take advantage of the cloud while keeping programming simple for developers.

We also took a good look at the OpenWhisk architecture and its components. Since constraints are critical to developing serverless applications, we also covered those in this chapter.

We wrapped up the chapter by comparing Apache OpenWhisk with Java EE to illustrate how serverless architecture has evolved.

CHAPTER 2

A Simple OpenWhisk Application

This chapter walks through developing a simple OpenWhisk application using the CLI and JavaScript as a programming language. The goal is to show you how serverless development works in action. For this purpose, we are going to create a simple contact form for a website using OpenWhisk.

Apache OpenWhisk is an open source project that is meant to be cloud-independent; you can adopt it without being constrained to one single vendor.

Let's assume you already have a static website and you want to add a contact form. The challenge here is that you cannot store contacts in your static website; you need some server logic to save them. Furthermore, you may want some logic to validate the data (since users can disable JavaScript in their browsers) and additionally receive email notification of what is going on on the website.

Since you cannot do this with just static HTML, this is a good use case for OpenWhisk: implementing simple external logic for an otherwise static website. For simplicity, our examples refer to the IBM Cloud, which offers OpenWhisk as its "Cloud Functions" service. However, the techniques here can be used with other cloud providers that support OpenWhisk.

The source code for the examples in this chapter is available on GitHub.

Getting Started

We're assuming that you already have a website, built with a static website generator. Even so, *not everything* on a website can be static. For example, you may need to allow your site's visitors to contact you. You may then want to store the contacts in a database to manage the data with a customer resource management (CRM) application.

Before serverless environments were available, the only way to enhance a static website with server-side logic was to provision a server (typically a VM), deploy a web server with server-side programming support, add a database, and finally implement the function you needed using a server-side programming language like PHP or Java. Today, you can keep your website mostly static and implement a contact form in a serverless environment, without the need for a VM, web server, or database.

The implementation that is shown in this chapter is not the best one possible. It is just a simple example to show you how to do it with OpenWhisk. In Chapters 4 and 5, we will rework the example to implement it in a more structured way.

The Bash CLI

Serverless development is primarily performed at the terminal using a CLI. We'll mostly use the Unix-based CLI and the command-line interpreter bash. In a sense, this is a Unix-centered book. Bash is available out of the box in any Linux-based environment and on macOS. It can also be installed on Windows.

You can use any Unix-like CLI, including Git for Windows. Note that, despite the name, it is not just Git: it includes a complete Unix-like environment, based on bash, ported to Windows. On more recent versions of Windows, you can instead install the Windows Subsystem for Linux (WSL), which includes a complete Linux distribution like Ubuntu, with bash of course. It is informally called Bash for Windows.

So, to follow along with these examples, you need to be familiar with bash and the command line. Here is an example of working with the CLI:

```
$ pwd                                                                    ❶
/Users/msciab
$ ls \                                                                   ❷
-l
# total 16                                                               ❸
-rw-r--r--@ 1 msciab  staff  1079 24 Feb 09:53 form.js
-rw-r--r--@ 1 msciab  staff    71 23 Feb 17:18 submit.js
```

❶ Type **`pwd`** at the command line.

❷ Type these two lines at the command line. While not strictly necessary here, this example is intended to show that long command lines may be split into multiple lines. In these cases, the lines that are continued should each end with a `\`.

❸ This is an example of the output you should expect.

The IBM Cloud

As I've mentioned a few times now, Apache OpenWhisk is an open source project and can be deployed in any cloud. You will see in Chapter 12 how to implement your environment on-premises or in the cloud of your choice. However, since OpenWhisk was initiated by IBM and then donated to the Apache Foundation, it is available ready to use in the IBM Cloud. The OpenWhisk environment is offered for free for development purposes, so the simplest way to get started is to create an IBM Cloud account.

At the time of writing, you can create an IBM Cloud account by going to *http://cloud.ibm.com* and looking for a button that says "Create an IBM Cloud account."

Fill in the form and then activate your account.

After registering, the next steps are as follows:

1. Log in to the IBM Cloud dashboard.
2. Download the CLI tool as shown in Figure 2-1.
3. Install the CLI tool using the installer for your platform.
4. Go to the terminal.
5. Log in to the IBM Cloud with `ibmcloud login`.
6. Install the Cloud Functions plugin with `ibmcloud plugin install cloud-functions`.
7. Target your organization and your workspace with `ibmcloud target -o <user name> -s dev`, where *`<username>`* is your username.

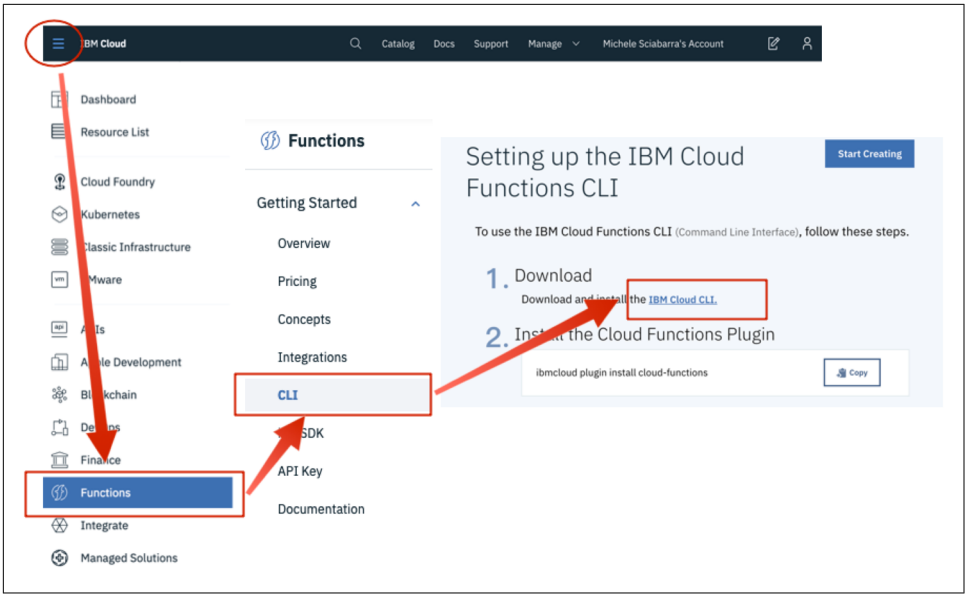


Figure 2-1. Downloading the CLI tools for OpenWhisk

In the IBM Cloud, when you register for a free account, your organization name is your email address, and there is only one space, `dev`. You can create more spaces, but you need to upgrade to a paid account.

To make sure you are ready, run the following command in the terminal and check the result. If your result matches this one, everything is configured correctly!

```
$ ibmcloud fn action invoke /whisk.system/utils/echo -p message hello --result
{
    "message": "hello"
}
```

In the rest of the book we use `wsk` on the command line to invoke OpenWhisk. This is the name of the CLI if you download it directly as described in "The wsk Command" on page 46. If you use the `ibmcloud` CLI, you have to use `ibmcloud fn` instead of `wsk`. To avoid confusion, we recommend using your shell alias functions and setting `alias wsk="ibmcloud fn"` in your shell initialization file, so you can use the command `wsk` to follow the examples in this book.

Creating a Simple Contact Form

Now we are going to create and deploy a simple contact form.

Open a terminal running bash. First we will create a package to group our code:

```
$ wsk package create contact
ok: created package contact
```

Now we are ready to start coding our application. In the serverless world you can write code and then deploy and execute it immediately. This immediacy is the essence of the platform.

We call the code we create *actions*. Actions are generally, at least in this chapter, single files. One exception is when we bundle an action file with some libraries we need, and we deploy them together. In such a case, the action will be deployed as a zip file, composed of many files.

Let's write our first action. The action is a simple HTML page returned by a JavaScript function:

```
function main() {
    return {
        body: `<html><head>
<link href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.0/css/bootstrap.min.css"
      rel="stylesheet" id="bootstrap-css">
</head><body>
 <div id="container">
  <div class="row">
   <div class="col-md-8 col-md-offset-2">
    <h4>Get in Touch</h4>
    <form method="POST" action="submit">
     <div class="form-group">
       <input type="text" name="name"
        class="form-control" placeholder="Name">
     </div>
     <div class="form-group">
        <input type="email" name="email"
         class="form-control" placeholder="E-mail">
     </div>
     <div class="form-group">
        <input type="tel" name="phone"
         class="form-control" placeholder="Phone">
     </div>
     <div class="form-group">
        <textarea  name="message" rows="3"
         class="form-control" placeholder="Message"
         ></textarea>
      </div>
```

```
      <button class="btn btn-default" type="submit" name="button">
        Send
      </button>
    </form>
   </div>
  </div>
 </div>
</body></html>`
    }
}
```

Note the general structure of the actions: the function is called `main`; it takes as an input a JavaScript `obj` and returns a JavaScript object.

Once you write the code, you can deploy it, then retrieve the actual URL to execute the action:

```
$ wsk action create contact/form form.js --web true                    ❶
ok: created action contact/form                                        ❷
$ wsk action get contact/form --url
ok: got action form
https://openwhisk.eu-gb.bluemix.net/api/v1\
/web/openwhisk%40example.com_dev/contact/form                          ❸
```

❶ Create the action and request web access for it.

❷ This message confirms the action was created.

❸ This is the URL where the action is available.

The actual URL returned will be different since it depends on your account and location.

As you will see later, not all the actions are accessible with a URL—only the "public-facing" ones. By default, actions are restricted and accessible only with an API key. We marked this action with the flag `--web true` because we want to access it with a URL and without authentication.

We can now test to see if the form was deployed correctly by opening the URL in a web browser. It should look like Figure 2-2.

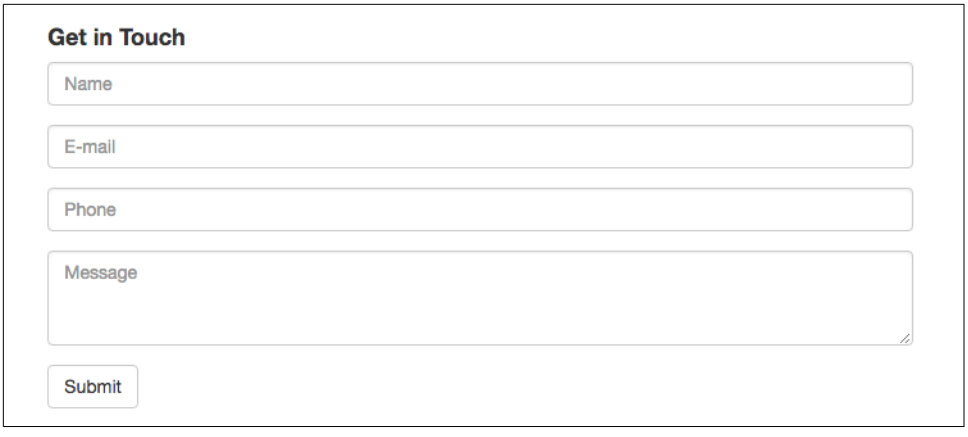


Figure 2-2. The contact form

If you click on the Submit button you will get an error, because it will try to submit the form to the action `contact/submit` that does not (yet) exist:

```
{
  "error": "The requested resource does not exist.",
  "code": 3850005
}
```

Form Validation

After testing, the next step is to validate the data the user submits before we store it in a database.

We will now create an action called `submit`, in the file *submit.js*. First, let's "wrap" our code in the `main` function:

```
function main(args) {
  message = []
  errors = []
  // TODO: <Form Validation>                    ❶
  // TODO: <Returning the Result>               ❷
}
```

❶ Insert the code for validating the form here.

❷ Insert the code for returning the result here.

Note that this action's `main` function has an explicit parameter, `args`. Let's see what happens when we submit the form.

We have a form with the fields `name`, `email`, `phone`, and `message`. On submit, the form data will be encoded and sent to OpenWhisk, which will decode it (all the details of the processing will be provided later in this chapter).

We use a JavaScript object to process a request using an action. Each key corresponds to a field of the form. This object is passed as the argument of the `main` function. So, our form fields are available to the `main` function as `args.name`, `args.email`, etc. We can now validate them.

Address Validation

First, let's check if the name is provided. Insert the following code snippet in place of the `// TODO: <Form Validation>` comment in the `submit.js` action:

```
// validate the name
if(args.name) {
 message.push("name: "+args.name)
} else {
 errors.push("No name provided")
}
```

Second, let's validate the email address, making sure it "looks like" an email. We use a regular expression for this (add this code below the code you just inserted in `action.js`):

```
// validate email
var re = /\S+@\S+\.\S+/;
if(args.email && re.test(args.email)) {
   message.push("email: "+args.email)
} else {
  errors.push("Email missing or incorrect.")
}
```

Third, let's validate the phone number, making sure it has enough digits (at least 10):

```
/// validate the phone
if(args.phone && args.phone.match(/\d/g).length >= 10) {
  message.push("phone: "+args.phone)
} else {
  errors.push("Phone number missing or incorrect.")
}
```

Finally, let's add the message text, if any:

```
/// add the message text, optional
if(args.message) {
  message.push("message:" +args.message)
}
```

Returning the Result

Once we have validated all the fields, we can return the result. There is now a bit of logic: we choose whether to return an error message or an acceptance message.

Insert the following code in place of the `// TODO: <Returning the Result>` comment in the `submit.js` action:

```
// complete the processing
if(errors.length) {
  var errs = "<ul><li>"+errors.join("</li><li>")+"</li></ul>"
  return {
    body: "<h1>Errors!</h1>"+
      data + errs +
      '<br><a href="javascript:window.history.back()">Back</a>'
   }
} else {
   var data = "<pre>"+message.join("\n")+"</pre>"
   // storing in the database
   // TODO: <Store the message in the database>                    ❶
   return {
     body: "<h1>Thank you!</h1>"+ data
   }
}
```

❶ Placeholder to insert code to save data in the database.

The action is not complete (we still need to store the data), but we can test this partial code with this command:

```
$ wsk action create contact/submit submit.js --web true
ok: created action contact/submit
```

If you now submit the form empty, you will see the message shown on the right in Figure 2-3. If you provide all the parameters correctly, you will instead see the message on the left.

Figure 2-3. Form messages

Saving Form Data

Now we can focus on database creation, to store our form data. Since we have an IBM Cloud account, the simplest option is to use the Cloudant database. For small amounts of data, it is free to use.

As shown in Figure 2-4, this is pretty simple to do:

1. From the menu at the top of the IBM Cloud window, select Catalog.
2. Search for the Cloudant NoSQL database.
3. Select Create to create a Lite instance.
4. Click Launch to access the administrative interface.
5. Click Create Database (not shown in the figure).
6. Finally, specify **`contact`** as the database name.

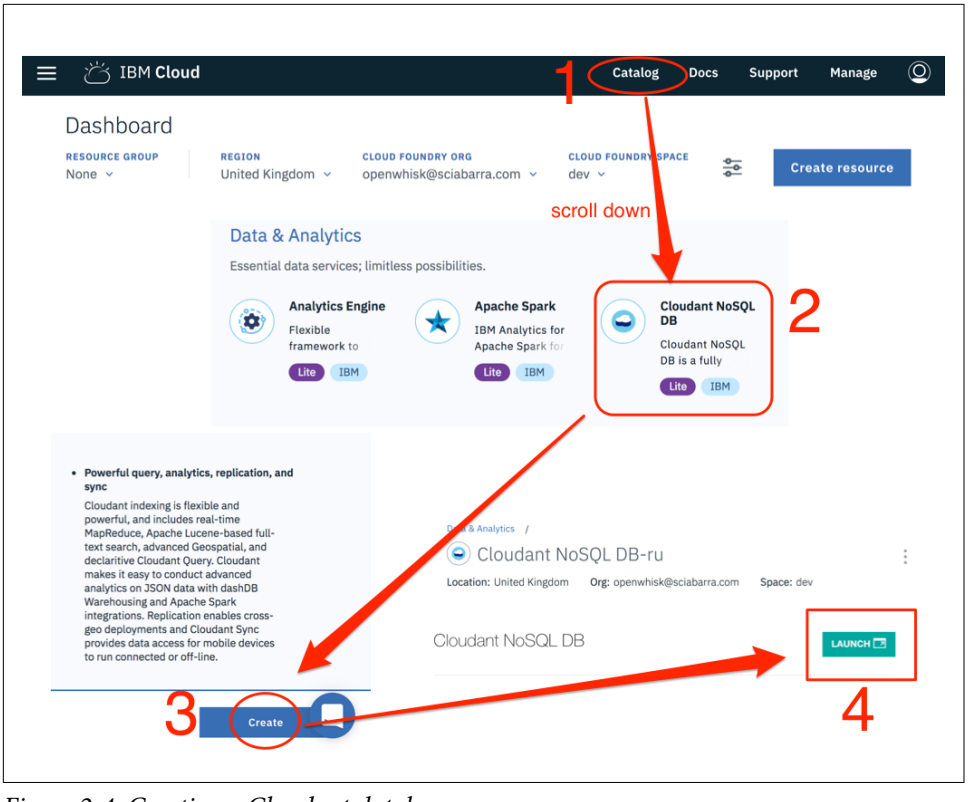


Figure 2-4. Creating a Cloudant database

Now we need to get the credentials to access the database.

As shown in Figure 2-5, in the Cloudant service, select "Service credentials" in the menu at the top left, then click "New credential," and finally select "View credentials." This will show a JSON file.

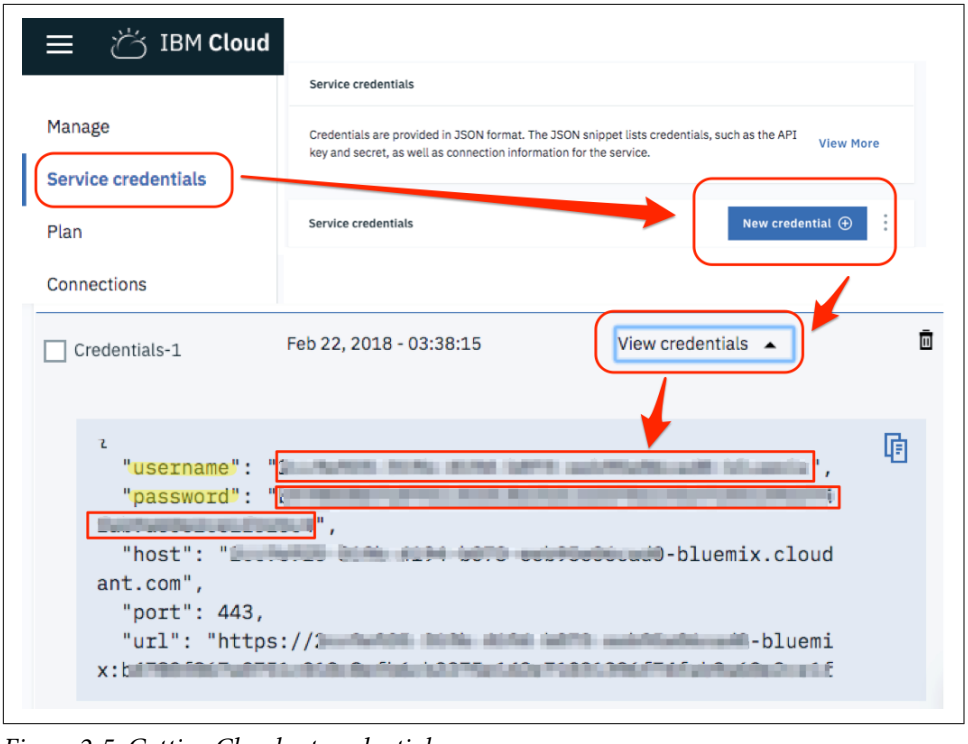


Figure 2-5. Getting Cloudant credentials

You need to extract the values in the fields `username` and `password`. You can now use these values at the command line to "bind" the database. This is what you have to type in the terminal:

```
$ export USER=XXXX                                                    ❶
$ export PASS=YYYY                                                    ❷
$ wsk package bind \                                                  ❸
   /whisk.system/cloudant contactdb \
  -p username $USER \
  -p password $PASS \
  -p host $USER.cloudant.com \
  -p dbname contactdb
ok: created binding contactdb
$ wsk action invoke contactdb/create-database -r                      ❹
{
    "ok": true
}
$ wsk package list                                                    ❺
packages
/openwhisk@example.com_dev/contactdb                  private
/openwhisk@example.com_dev/contact                    private
```

❶ Copy here the username.

❷ Copy here the password.

❸ Bind the `cloudant` package (one of a set of generic packages OpenWhisk provides with the specified credentials).

❹ Create the actual database.

❺ Make sure you have two packages.

In the IBM Cloud, it is possible to bind the Cloudant package automatically with the `reload` command. This will create a package with the credentials automatically configured.

After binding you will have a new package with parameters allowing you to read and write in your database. OpenWhisk provides a set of generic packages. We used the command `bind` to make it specific to write in our database giving username and password. This package provides many actions, which you can see by listing them (not all the actions are shown here):

```
$ wsk action list contactdb
/whisk.system/cloudant/write                    private nodejs:6
/whisk.system/cloudant/read                     private nodejs:6
...
```

Creating a package gives you access to multiple databases. Since we need just one, we specified the default database we want to use (using the parameter `dbname` in the `bind` command), then we explicitly invoked the `create-database` command to be sure that database was created.

Invoking Actions

The actions we've created so far have been web actions (you may have noted the `--web true` flag in the `action create` commands). In general, web actions can be invoked directly, navigating to web URLs or submitting web forms.

However, not all actions can be invoked straight from the web. Many of them execute internal processing and are activated only through specific APIs. We will also invoke other actions that are *not* web actions.

To read and write to a database, we have to use the `wsk action invoke` command, passing parameters with the `--param` option and data in JSON format as a command-line argument.

Let see how it works by invoking the action `write`, as follows:

```
$ wsk action invoke contactdb/write \
  --blocking --result --param dbname contactdb \
  --param doc '{"name":"Michele Sciabarra", "email": "michele@sciabarra.com"}'
{
    "id": "fad432c6ea1145e71e99053c0d811475",
    "ok": true,
    "rev": "1-d14ba6a37cfdf34a8b3bb49dd3c0e22a"
}
```

Here, we're using the `--blocking` option to wait for the result and the `--result` option to print the final result. Those options will be discussed in detail in the next chapter. However, note that in this case the `--blocking` is actually redundant; `--result` implies `--blocking`, so when using `--result` it is not required (it's included only for illustration purposes). OpenWhisk provides a set of generic packages. We used the `bind` command to create a copy that also provides additional parameters to let you read and write in a specific database.

The database replied with a confirmation and some extra information. Most notably, the result includes the id of the new record that we can use later to retrieve the value.

We can now check the database's user interface to see the data we just stored in it. Select the database and then the ID, as shown in Figure 2-6, and you should see the data we just inserted in the database in JSON format. As you can see, to write data in the database you just need a JSON object. So far we've written data using the command line. Now let's look at how we can do it in code.

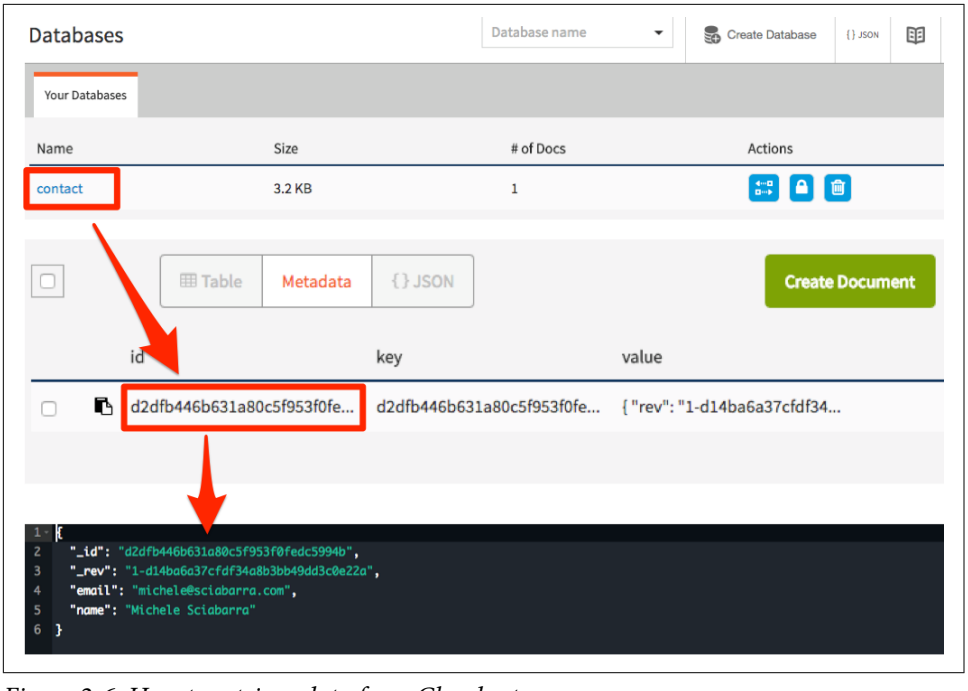


Figure 2-6. How to retrieve data from Cloudant

Storing in the Database

OpenWhisk includes an API you can use to interact with the rest of the system. We used the database as a package, so we now have an action to write in the database. To use it we need to invoke the action from another action. Let's see how.

First we need this line:

```
var openwhisk = require('openwhisk')
```

to access to the `openwhisk` API. This code is the entry point to interact with OpenWhisk from within an action.

Now we can define the function `save` using the OpenWhisk API to invoke the action to write in the database. The following code should be placed at the beginning of *submit.js*:

```
var openwhisk = require('openwhisk')

function save(doc)  {
  var ow = openwhisk()
  return ow.actions.invoke({
    "name": "contactdb/write",
    "params": {
       "dbname": "contactdb",
```

```
        "doc": doc
      }
    })
}
```

Currently, you need to add `var ow = openwhisk()` inside the body of the `main` function. This is a documented issue: environment variables used to link the library to the rest of the system are available only within the `main` function or any function called by it, so you cannot initialize the variable outside (as you may be tempted to do).

Once the `save` function is available, you can use it to save data just by creating an appropriate JSON object and passing it as a parameter. In *submit.js*, replace the placeholder line `// TODO: <Store the message in the database>` with the following code:

```
save({
 "name": args.name,
  "email": args.email,
  "phone": args.phone,
  "message": args.message
})
```

We "extracted" the arguments instead of just saving the whole `args` object because it includes other information we do not want to keep.

The code is now complete. We can deploy `submit` as a web action to be executed when the user submits the form:

```
$ wsk action update contact/submit submit.js --web true
ok: updated action contact/submit
```

We can now manually test the action from the command line to see if it writes in the database:

```
$ wsk action invoke contact/submit -p name Michele \
-p email michele@sciabarra.com -p phone 1234567890 -r
{
    "body": "<h1>Thank you!</h1><pre>name: Michele
email: michele@sciabarra.com
phone: 1234567890</pre>"
}
```

And we can try our form and make sure the data is in the database, as shown in Figure 2-7.

In general, automated testing is used to test serverless code. Here, we use manual testing to help you understand the system and to illustrate the manual testing process.

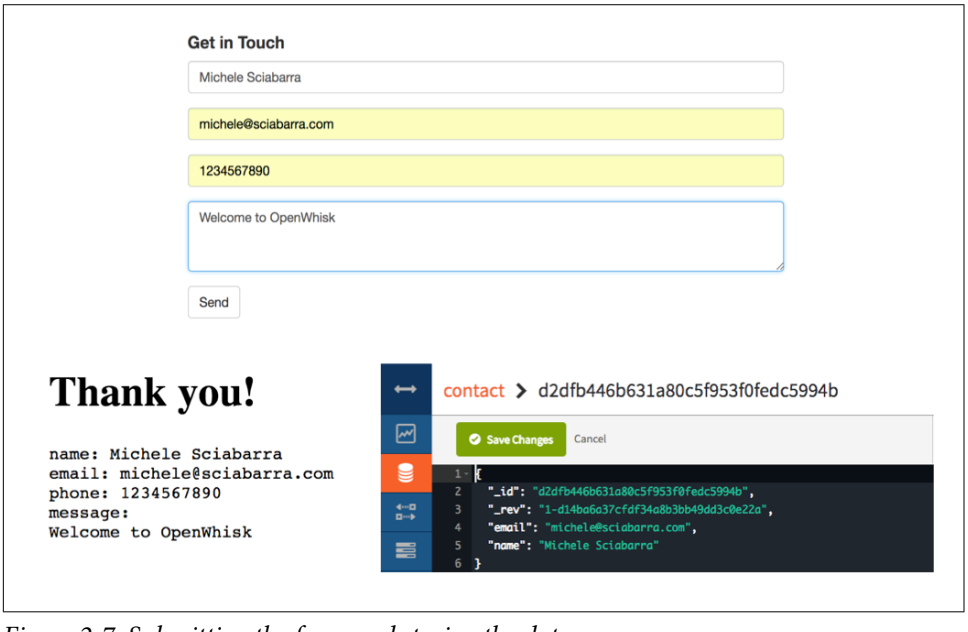


Figure 2-7. Submitting the form and storing the data

Sending an Email

Now let's add a feature that shows how the system interacts with other systems. We will set up the system to send an email when someone uses our contact form (for both complete and incomplete submissions).

We are going to use the Mailgun service for this purpose. It provides free accounts, allowing a limited number of emails to be sent only to selected and validated addresses—just what we need.

Sending email is usually done with the SMTP protocol, but since email is generally used to send spam in most cloud services, cloud providers usually block SMTP ports. Since this is also the case in the IBM Cloud, for our example we use a third-party service for sending email that provides an HTTP API and a JavaScript library to use it.

Configuring Mailgun

To use Mailgun go to `www.mailgun.com` and click the Sign Up button at the top right to register for a free account. Once registered, to retrieve credentials for sending an email (Figure 2-8) you need to:

1. Log in to Mailgun.
2. Scroll down until you find the Sandbox Domains.
3. Click on Authorized Recipients
4. Invite the recipient.
5. Click on the link in the email received.

Once you've done that, you'll need the following information, as shown in the figure, to write your script:

1. Sandbox domain
2. Private API key
3. Authorized recipient address

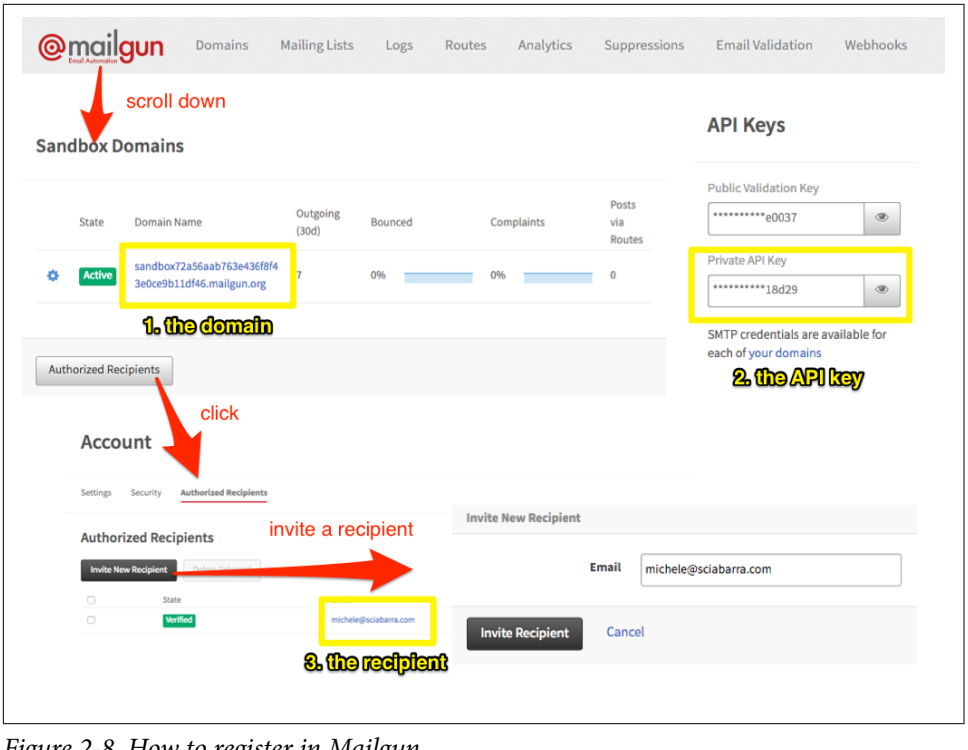


Figure 2-8. How to register in Mailgun

Writing an Action to Send Email

Now we can build an action to send an email. The purpose of this example is to show you how to create more complex actions involving third-party services and libraries.

This action is a bit more complicated than the actions we have seen before, because we need to use and install an external library and deploy it with the action code. We were able to skip the integration of libraries in the preceding steps only because the `cloudant` package was included in the runtime, since it is part of the standard OpenWhisk deployment.

Let's start by creating a folder and importing the library `mailgun-js` with the `npm` tool distributed with Node.js:

```
$ mkdir sendmail
$ cd sendmail
$ npm init -y
$ npm install --save mailgun-js
```

Now we can write a simple action that can send an email and place it in *sendmail/index.js*. Substitute in the information you collected in the previous section when registering with Mailgun:

```
var mailgun = require("mailgun.js")
var mg = mailgun.client({username: 'api',
        key: '<YOUR-PRIVATE-API-KEY>'})                          ❶
function main(args) {
  return mg.messages.create(
          '<YOUR-SANDBOX-DOMAIN>.mailgun.org', {                 ❷
    from: "<YOUR-RECIPIENT-EMAIL>",                              ❸
    to: ["<YOUR-RECIPIENT-EMAIL>"],                              ❸
    subject: "[Contact Form]",
    html: args.body
  }).then(function(msg) {
     console.log(msg);
     return args;
  }).catch(function(err) {
     console.log(err);
     return args;
  })
}
```

❶ Replace **`<YOUR-PRIVATE-API-KEY>`** with the private API key.

❷ Replace **`<YOUR-SANDBOX-DOMAIN>.mailgun.org`** with the sandbox domain.

❸ Replace **`<YOUR-RECIPIENT-EMAIL>`** with the one used both as the sender and recipient.

For simplicity, we've placed the keys within the script, but this is not a recommended practice. You will see later how to define parameters for a package and use them for each action, so you do not have to store keys in the code.

We can now deploy the action. Since this time the action is not a single file, because it includes libraries, we have to package everything in a zip file:

```
$ zip ../sendmail.zip -q -r *
$ wsk action update contact/sendmail ../mailgun.zip --kind nodejs:6
ok: updated action contact/sendmail
```

We are using `update` here even though the action was not present before. `update` will create the action if it does not exist, so by using it we avoid the error `create` would have given if the action were already there. Also note that because we placed the action in a zip file, we need to specify its type; the command line cannot deduce it from the filename.

We can now test the action by invoking it directly:

```
$ wsk action invoke contact/sendmail -p body "<h1>Hello</h1>" -r
{
    "body": "<h1>Hello</h1>"
}
```

The action acts as a filter, accepting the input and copying it in the output. We will leverage this behavior to chain the action to `submit`. When invoked, it will return an output, but also send an email as a side effect (see Figure 2-9).

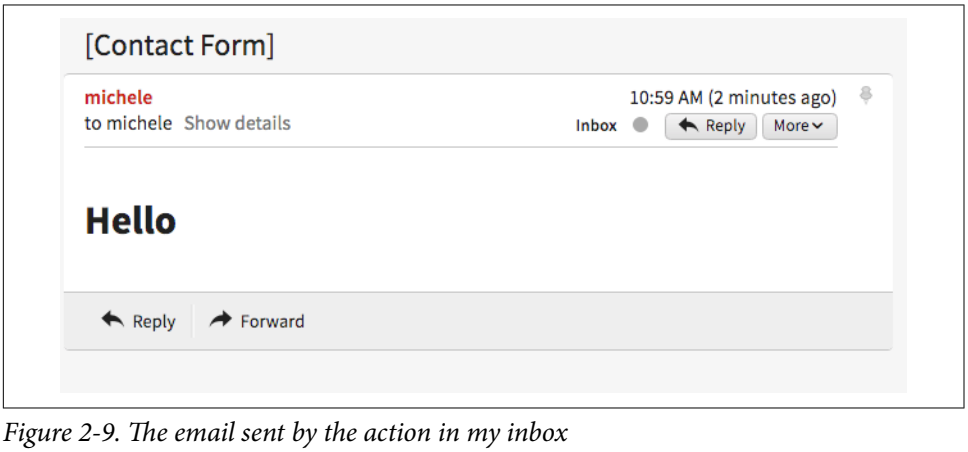


Figure 2-9. The email sent by the action in my inbox

Creating an Action Sequence

So far we have developed an action that can send an email as a standalone action. But we designed it to take the output of the `submit` action and return it as is so we can create a pipeline of actions, similar to Unix shell processing, where the output of a command is used as an input for another command.

We are going to take the output of the `sendmail` action, use it as the input for the `submit` action, and then return it as the final result of the email submission.

Note that it will send emails for every submission, even for incorrect inputs, so we will know if someone is trying to use the form without providing all the information. But we will only store the fully validated data in the database. Let's create this pipeline, called a *sequence* in OpenWhisk, and then test it:

```
$ wsk action create contact/submit-sendmail \
  --sequence contact/submit,contact/sendmail \
  --web true
ok: updated action contact/submit-sendmail

$ wsk action invoke contact/submit-sendmail -p name Michele -r
{
    "body": "<h1>Errors!</h1><pre>name: Michele</pre><ul>\
<li>Email missing or incorrect.</li>\
<li>Phone number missing or incorrect.</li></ul>\
<br><a href=\"javascript:window.history.back()\">Back</a>"
}
```

As a result, we should receive an email for each attempt at submitting the form (both the complete and incomplete attempts).

Now we can complete the whole process including storing the data in the database and sending the email. But the form should activate our sequence, not just the submission action. The simplest way to activate the sequence is to edit the *form.js* file, replacing the action we are invoking in the form. Make the following change in *form.js*:

```
-<form method="POST" action="submit">
+<form method="POST" action="submit-sendmail">
```

The change we are making here is in the *patch* format. We won't get into too many details here, but in general the prefix - means remove this line, while the prefix + means add this line. We will use this notation again later in the book.

If we now update the action and try it, we should receive an email for each form submission, but the form will be stored in the database only when the data is complete.

Summary

In this chapter, we started from scratch with OpenWhisk and created an IBM Cloud account. Then we downloaded and installed the command-line interface, its primary access mechanism. We used that account to write a simple application and to explore some of the mechanics of serverless programming.

We created a simple HTML form in OpenWhisk and then implemented basic form validation logic, storing the results in a NoSQL database (Cloudant). Finally, we connected to a third-party service (Mailgun) to send an email.

Congratulations on your first exposure to OpenWhisk in practice!

CHAPTER 3

The OpenWhisk CLI and JavaScript API

OpenWhisk applications are made up of *entities* you can manipulate with the command line or programmatically. The CLI uses the command `wsk`, which can be used interactively or by writing automated scripts. You can also use JavaScript, using an API crafted explicitly for OpenWhisk. These are both external interfaces to the REST API OpenWhisk exposes. Since they are two different aspects of the same thing, they are both covered in this chapter.

Before we get into design patterns in the next few chapters, first you need to learn about the OpenWhisk API. This API is critical to writing applications that leverage specific OpenWhisk features.

We already covered the fundamentals of OpenWhisk in the example in Chapter 2, but let's recap a few things here:

- Packages are used to group actions together, share parameters, and annotations, etc., and they also provide a base URL that can be used by web applications.
- Actions are the building blocks of an OpenWhisk application, and can be written in one of the programming languages supported by OpenWhisk; they receive input and provide an output, both in JSON format.
- Actions can be interconnected, where the output of one action becomes the input of another, thus creating a sequence.
- Triggers are similar to actions but are used through rules to activate multiple actions.
- Rules associate triggers with actions, so when you fire a trigger, all its actions are invoked.

- Feeds are specially crafted actions with a well-defined pattern; they connect events provided by a package with triggers defined by a consumer.

The source code for the examples in this chapter related to the CLI is available in the GitHub repository, as is the source code for the examples related to the API.

The wsk Command

Let's begin with the CLI, using the command `wsk`. This command can be downloaded in precompiled binary format for many platforms from the releases page of the repository.

The `wsk` command is composed of many commands, each with many subcommands. The general format is this:

```
wsk <COMMAND> <SUBCOMMAND> <PARAMETERS> <FLAGS>
```

<PARAMETERS> and *<FLAGS>* are different for each *<SUBCOMMAND>*, and for each *<COMMAND>* there are many subcommands.

The CLI itself is self-documenting and provides help when you do not feed enough parameters to it. Typing `wsk` will get you a list of the main commands. If you type the `wsk` command then a subcommand, you will get help for that subcommand. For example:

```
$ wsk
Available Commands:
  action      work with actions
  activation  work with activations
  package     work with packages
  rule        work with rules
  trigger     work with triggers
  property    work with whisk properties
  namespace   work with namespaces
  list        list entities in the current namespace
$ wsk action
  create      create a new action                          ❶
  update      update an existing action                    ❷
  invoke      invoke action
  get         get action                                   ❸
  delete      delete action                                ❹
  list        list all action                              ❺
```

❶ `create`: available for actions, packages, rules, and triggers.

❷ `update`: available for actions, packages, rules, and triggers.

❸ `get`: available for actions, packages, rules, and triggers, and also for activations, namespaces, and properties.

❹ `delete`: available for actions, packages, rules, and triggers.

❺ `list`: available for actions, packages, rules, and triggers, and also for namespaces, and as a top-level command to list everything.

Remembering that the commands are the entities and the subcommands are CRUD (create, retrieve, update, delete) will help you remember how the `wsk` command works. Of course, there are individual cases that differ, but we'll cover those as we go.

Subcommands also have flags. As we discuss the subcommands, we will cover some of their flags as they come up.

Configuring the wsk Command

The `wsk` command has a configuration, in the form of a set of properties you can set; those properties are the credentials to access an OpenWhisk environment. When you use the IBM Cloud, those properties are actually set for you by the `ibmcloud` main command. If you have a different OpenWhisk deployment, you may need to change the properties manually.

You can see the currently configured properties with:

```
$ wsk property get
Client key
whisk auth            xxxxx:YYYYY                    ❶
whisk API host        openwhisk.eu-gb.bluemix.net    ❷
whisk API version     v1
whisk namespace       _                              ❸
whisk CLI version     2017-11-15T19:55:32+00:00
whisk API build       2018-02-28T23:44:25Z
whisk API build number    whisk-build-7922
```

❶ API authentication key (replaced in the example with `xxxxx:YYYYY`)—your own will be different.

❷ OpenWhisk host to which you connect to control OpenWhisk with the CLI—depends on your location.

❸ Current namespace (the value _ is shorthand for your default namespace).

If you install a local OpenWhisk environment, you need to set these properties manually using `wsk property set`. In particular, you need to set the `whisk auth` and `whisk host` properties with the values provided by your local installation.

OpenWhisk Entity Names

Let's see how we name things. As you may already know, in OpenWhisk we have the following entities with names: packages, actions, sequences, triggers, rules, and feeds. We have to use precise naming conventions for each, and they are reflected in the structure of the URLs used to invoke the various elements of OpenWhisk.

The general structure for the entity URL is "namespace/package/entity", where "package" is optional for all the entities, or not required for some entities. As a result, all the entities are placed in some namespace. Actually, a namespace is assigned by system administrators to users when they are given access to the system. A namespace acts a workspace and provides a base URL for all the entities the user can create. The credentials that are given to a user allow access to entities placed under a namespace.

For example, when I registered with the IBM Cloud, my namespace was `/openwhisk@example.com_dev/`: my username followed by `_dev` (for development).

You can create a namespace in an OpenWhisk installation if you are authorized, or the system administrator can create one for you.

Under a namespace you can create triggers, rules, actions, and packages. They will have names like `/openwhisk@example.com_dev/a-trigger`, `/openwhisk@example.com_dev/a-rule`, `/openwhisk@example.com_dev/a-package`, and `/openwhisk@example.com_dev/an-action`.

When you create a package, you can include its actions and feeds. For example, in the package `a-package` you can have `/openwhisk@example.com_dev/a-package/another-action` and `/openwhisk@example.com_dev/a-package/a-feed`.

To recap:

- The general format for entities is `/<namespace>/<package>/<entity>`, but it can be reduced to `/<namespace>/<entity>`.
- Under a namespace you can create triggers, rules, packages, and actions.
- Under a package you can create actions, but not triggers, rules, or other packages.

Most of the time you do not need to specify the namespace. If you specify an action as a relative action (not starting with `/`) it will be placed in your current namespace. Note that the special namespace `_` means "your current namespace" and the full namespace name automatically replaces it.

Defining Packages

In OpenWhisk, all the entities are grouped in a namespace. You can put actions directly under a namespace, or you can create a package instead, and put actions under the package. Packages are used to:

- Group related actions together, to reuse them and share them with others
- Share parameters, annotations, etc.
- Provide a URL for those related actions (useful for actions that refer to each other, like in web applications)

Let's create a package called `sample` and provide a parameter, as follows:

```
$ wsk package create sample -p email michele@sciabarra.com
ok: created package sample
```

Keep in mind that when you set parameters for a package, those parameters are available to all the actions in the package. We will use this feature frequently in the examples.

You can `list` a package, `get` information from it, `update` it (e.g., with different parameters), and finally `delete` it:

```
$ wsk package list
packages
/openwhisk@example.com_dev/sample                                      private
/openwhisk@example.com_dev/contact                                     private
/openwhisk@example.com_dev/contactdb                                   private
$ wsk package update sample -p email openwhisk@example.com
ok: updated package sample
$ wsk package get sample -s                                                    ❶
package /openwhisk@example.com_dev/sample: Returns a result based on parameter
   email (parameters: *email)
$ wsk package delete sample
ok: deleted package sample
```

❶ Here we used the parameter `-s` to summarize information from the package.

Package Binding

Now let's discuss another function of packages: binding. OpenWhisk allows you to import (or *bind*) third-party packages to your namespace to customize it.

Keep in mind that user credentials allow access to all the resources under a namespace. Binding a package therefore makes it accessible to the other actions in the namespace.

To better understand how binding works, let's look at the packages for databases and message queues available in the IBM Cloud.

A package for Cloudant is available, and all we need to do to use it is to bind it. Binding has the purpose of adding parameters to a package with a new name to make access easier. In the following example, we use the configuration file *cloudant.json* (we covered how to retrieve configuration files in Chapter 2):

```
$ wsk package list /whisk.system
packages                                                              ❶
/whisk.system/cloudant                                        shared
/whisk.system/websocket                                       shared
/whisk.system/alarms                                          shared
/whisk.system/messaging                                       shared
$ wsk package get /whisk.system/cloudant  -s | head -2                ❷
package /whisk.system/cloudant: Cloudant database service
   (parameters: *apihost, *bluemixServiceName,
    *dbname, *host, overwrite, *password, *username)                  ❸
$ wsk package bind /whisk.system/cloudant patterndb  \                ❹
    -P cloudant.json -p dbname pattern                                ❺
ok: created binding contactdb
```

❶ Here, we list the packages available in the IBM Cloud (the output has been edited and shortened for clarity).

❷ Here, we inspect the Cloudant package (only the first two lines are shown).

❸ Note here the required parameters to use the database.

❹ Here, we create the binding to make the database accessible.

❺ We are using the file *cloudant.json* to specify the `host`, `username`, and `password`, and using `dbname` on the command line.

Two common flags, available also for actions, feeds, and triggers, are `-p` and `-P`. With `-p <name> <value>` you can specify a parameter named *<name>* with value *<value>*. With `-P` you can put some parameters in a JSON file, which is assumed to be a map. See Figure 4-6 for an example of this format.

Creating Actions

The purpose of the `wsk action` command is to manipulate actions using CRUD operations and the subcommands. Let's illustrate this using a simple `now` action:

```
function main(args) {
  return { body: Date() }
}
```

Now, if we want to deploy this simple action in the package `basics` we use the following:

```
$ wsk package update basics                                    ❶
ok: updated package basics
$ wsk action create basics/now basics/now.js                   ❷
ok: created action basics/now
```

❶ Ensures we have a `basics` package.

❷ Creates the action from the file stored in *basics/now.js*.

Now that the action has been deployed, we can invoke it. The simplest way is to call it as follows:

```
$ wsk action invoke basics/now
ok: invoked /_/basics/now with id fec047bc81ff40bc8047bc81ff10bc85
```

Wait a minute… where is the result? If you remember correctly, actions in OpenWhisk are by default asynchronous, so we usually get an id (called the activation ID) to retrieve the result after the action completed. We'll discuss this in more detail in the next section.

If we instead want to see the result immediately, we can provide the flag `-r` or `--result`, which blocks until we get an answer:

```
$ wsk action invoke basics/now -r
{
    "body": "Thu Mar 15 2018 14:24:39 GMT+0000 (UTC)"
}
```

But what if we want to access that action from the web? To do that, we can retrieve a URL with `get` and `--url`. If we leave out the `--url` we get a complete description of the action in JSON format:

```
$ wsk action get basics/now --url
https://openwhisk.eu-gb.bluemix.net/api/v1\
/namespaces/openwhisk@example.com_dev/actions/basics/now
$ wsk action get basics/now
{
    "namespace": "openwhisk@example.com_dev/basics",
    "name": "now",
    "version": "0.0.1",
    "exec": {
        "kind": "nodejs:6",
        "binary": false
    },
    "annotations": [
        {
            "key": "exec",
            "value": "nodejs:6"
        }
    ],
    "limits": {
        "timeout": 60000,
        "memory": 256,
        "logs": 10
    },
    "publish": false
}
```

But if we try to use the URL to run the action we may get a nasty surprise:

```
$ curl https://openwhisk.eu-gb.bluemix.net/api/v1/\
namespaces/openwhisk@example.com_dev/actions/basics/now
{"error":"The resource requires authentication,\
 which was not supplied with the request"
 "code":9814}
```

While all the actions (and everything else) in OpenWhisk are accessible with a REST API, they are protected and not accessible without authentication.

However, it is possible to mark an action as publicly accessible with the `--web true` flag when creating or updating it. These actions are called *web actions*. A web action produces web output, allowing you to view it with a browser. While there are some constraints on web actions, as we'll discuss a little later, for now it's enough to know that you must have a `body` property to render an HTML page.

To change our action to a web action, we will use the `update` command. Then we can immediately retrieve its URL and invoke it directly:

```
$ wsk action update basics/now --web true
ok: updated action basics/now
$ curl $(wsk action get basics/now --url | tail -1)
Thu Mar 15 2018 14:46:56 GMT+0000 (UTC)
```

Now that we've covered `create` and `update`, let's wrap up by demonstrating the `list` and `delete` commands:

```
$ wsk action list basics
actions
/openwhisk@example.com_dev/basics/now                    private nodejs:6
$ wsk action delete basics/now
ok: deleted action basics/now
$ wsk action list basics
actions
```

Chaining Sequences of Actions

An essential feature of OpenWhisk is the ability to chain actions in sequences, creating actions that use, as an input, the output of another action, as shown in Figure 3-1.

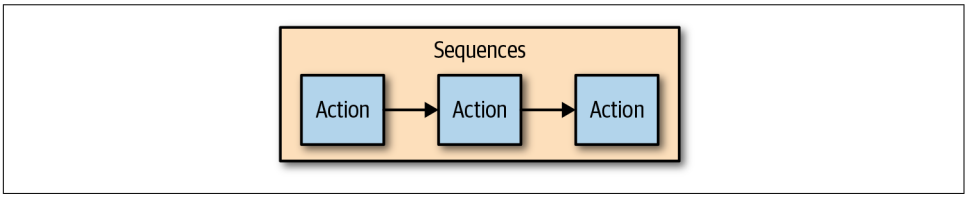


Figure 3-1. Actions chained in a sequence

Let's go over a simple example implementing a word count application separated into two actions to demonstrate how to chain sequences of actions. The first action splits the input, which is supposed to be words in a text file, while the second retrieves the words and produces a map as a result. In the map, each word is then shown with its frequency.

Let's start with the first action, in the file *split.js*:

```
function main(args) {
    let words = args.text.split(' ')
    return {
        "words": words
    }
}
```

You can deploy and test the `split` action by feeding it a simple string:

```
$ wsk action update basics/split basics/split.js
ok: updated action basics/split
$ wsk action invoke basics/split \
  -p text "the pen is on the table" -r \
  | tee save.json                                          ❶
{
    "words": [
        "the",
        "pen",
```

```
        "is",
        "on",
        "the",
        "table"
    ]
}
```

❶ Note here that we are saving the output in a file called *save.json*.

Let's do the second step. Here are the contents of the *count.js* file:

```
function main(args) {
    let words = args.words
    let map = {}
    let n = 0
    for(word of words) {
       n = map[word]
       map[word] = n ? n+1 : 1
    }
    return map
}
```

We can now deploy the `count` action and check the result, feeding the output of the first action as input:

```
$ wsk action update basics/count count.js
ok: updated action basics/count
$ wsk action invoke basics/count -P save.json -r
{
    "is": 1,
    "on": 1,
    "pen": 1,
    "table": 1,
    "the": 2
}
```

Now we have two actions. Since the second can take the output of the first as input, we can create a sequence:

```
$ wsk action update basics/wordcount \
  --sequence basics/split,basics/count                                    ❶
```

❶ Note here that we are specifying a comma-separated list of existing action names.

The sequence can now be invoked as a single action; we feed the text input and see the result:

```
$ wsk action invoke basics/wordcount -r -p text  \
"can you can a can as a canner can can a can"
{
    "a": 3,
    "as": 1,
    "can": 6,
```

```
    "canner": 1,
    "you": 1
}
```

Including Some Code of Your Own as a Library

In essence, an action is just a single JavaScript file. But it is common to want to share code or to use it in actions. The best way to handle this situation is to have a library of code that you deploy with your actions.

Let's consider a couple of utility functions that format the date in the standard format "YYYY/MM/DD" and the time in the standard format "HH:MM:SS":

```
function fmtDate(d) {
    let month = '' + (d.getMonth() + 1),
        day = '' + d.getDate(),
        year = d.getFullYear();
    if (month.length < 2) month = '0' + month;
    if (day.length < 2) day = '0' + day;
    return year + "/" + month + "/" + day
}

function fmtTime(d) {
    let hour =  ''+ d.getHours(),
        minute = '' + d.getMinutes()
        second = '' + d.getSeconds()

    if(hour.length < 2) hour = "0"+hour
    if(minute.length < 2) minute = "0"+minute
    if(second.length <2 ) second = "0"+second

    return hour + ":" + minute + ":" + second
}
```

Of course, you could copy this code into each action. But this could become tedious and would be difficult to maintain, because if you change the functions you will have to update all the files that use those functions.

Fortunately there's a better way. Since actions are executed using Node.js, you can use the standard export/require mechanism. As a convention, we are going to place our shared code in a subdirectory named *lib*, and we'll treat it as modules.

So, place the preceding code in a file called *lib/datetime.js* and add the following code at the end:

```
module.exports = {
    fmtTime: fmtTime,
    fmtDate: fmtDate
}
```

Now you can use the two functions in one action. For example, let's consider an action named `clock` that returns the date if invoked with `date=true`, the time if invoked with `time=true`, or both if the `date` and `time` parameters are specified.

Our action starts by requiring the library with the following:

```
var dt = require("./lib/datetime.js")
```

This way we can access the two functions as `dt.fmtDate` and `dt.fmtTime`. Using those functions, we can easily write the main body, *clock.js*, as:

```
function main(args) {
  let now = args.millis ? new Date(args.millis) : new Date()
  let res = " "
  if(args.date)
    res = dt.fmtDate(now) + res
  if(args.time)
    res = res + dt.fmtTime(now)
  return {
    body: res
  }
}
exports.main = main
```

Now we have to deploy the action and include the library. How can we do this in OpenWhisk?

The solution is to deploy not a single file but a zip file that includes all the files we want to run as a single action. When deploying multiple files, you need either to put your code in an *index.js* file or provide a *package.json* file saying which one is the main file.

Let's perform this procedure with the following commands, creating a *package.json* inline, then a zip file, and finally deploying it all. The final content of the *clock.zip* file will be as follows:

```
├── clock.js
├── lib
│   └── datetime.js
└── package.json
```

Let's build and deploy it:

```
$ cd basics
$ echo '{"main":"clock.js"}' >package.json                    ❶
$ zip -r clock.zip clock.js package.json lib                  ❷
$ wsk action update basics/clock clock.zip \
  --kind nodejs:6                                             ❸
ok: updated action basics/clock
```

❶ Create a simple *package.json* on the fly.

❷ Create a zip file with subdirectories (this requires the `-r` switch).

❸ Deploy the action, specifying that the runtime we want to use is `nodejs:6`.

When we deploy an action as a zip file, we have to specify the runtime to use with `--kind nodejs:6`, because the system is unable to determine this from the filename.

Let's test it:

```
$ wsk action invoke basics/clock -p date true -r
{
    "body": "2018/03/18 "
}
$ wsk action invoke basics/clock -p time true -r
{
    "body": " 15:40:42"
}
```

Inspecting Activations

As we saw earlier, when we invoke an action without waiting for the result, we receive an invocation idea. This fact brings us to the topic of this section: the use of the subcommand `wsk activation` to manage the results of invocations.

To explore it, let's create a simple *echo.js* file:

```
function main(args) {
 console.log(args)
 return args
}
```

Now let's deploy and invoke it (with the parameter `hello=world`) to get the activation ID (for more on activation and invocation IDs, see Chapter 1):

```
$ wsk action create basics/echo echo.js
ok: created action basics/echo
$ wsk action invoke basics/echo -p hello world
ok: invoked /_/basics/echo with id 82deb0ec37524a9e9eb0ec37525a9ef1
```

As explained in Chapter 1, when actions are invoked, they are identified by an activation ID that can be used to save and retrieve results and logs from the database. We can use this activation ID with the option `result` to get the results, and `logs` to get the logs:

```
$ ID=$(wsk action invoke basics/echo -p hello world \
  | awk '{ print $6}')
$ wsk activation result $ID
```

```
{
    "hello": "world"
}
$ wsk activation logs $ID
2018-03-15T18:17:36.551486467Z stdout: { hello: 'world' }
```

You can also use `wsk result --last` to get the result of the last invoked action.

The `activation` subcommand has a few other helpful options. One is `list`. It will return a list of *all* the activations in chronological order. Since the list can be very long, it's useful to use the option `--limit` *<n>* to see only the latest *<n>*:

```
$ wsk activation list --limit 4
activations
82deb0ec37524a9e9eb0ec37525a9ef1 echo
219bacbdb838449d9bacbdb838149de2 echo
1b75cd02fd1f4782b5cd02fd1f078284 echo
6fa117115aa74f90a117115aa7cf90e0 now
```

Another handy option (probably the most useful) is `poll`. With this option, you can continuously display logs for actions as they occur. This lets you monitor what is going on in the remote serverless system when you are debugging or doing other similar tasks.

You can poll for just one action or for all the actions at the same time.

Managing Triggers and Rules

Now let's see how to create a trigger. A trigger is merely a name for an arbitrary event. By itself, a trigger can do nothing. However with the help of rules, it becomes useful, because when a trigger is activated, it invokes all the associated rules, as shown in Figure 3-2.

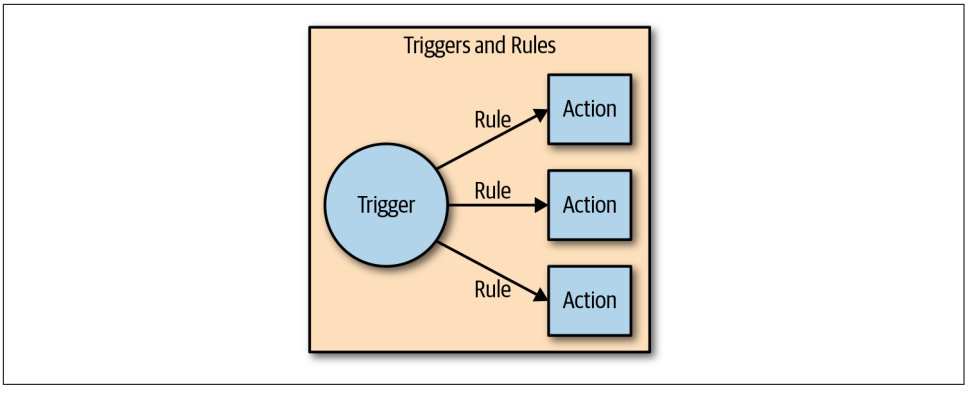


Figure 3-2. Triggers and rules

Let's look at a simple logging example to see how triggers work. After that, we'll get into how rules work.

First let's prepare our example, deploying a simple `log` action that logs its name:

```
function main(args) {
 console.log(args.name)
 return {}
}
```

Then we deploy it twice, with two different names:

```
$ wsk action update basics/log-alpha -p name alpha basics/log.js
ok: updated action basics/log-alpha
$ wsk action update basics/log-beta -p name beta basics/log.js
ok: updated action basics/log-alpha
```

By themselves, those actions do nothing except leave a trace of their activation in the logs:

```
$ wsk action invoke basics/log-alpha                                    ❶
ok: invoked /_/basics/log-alpha with id 320b50d841064d0b8b50d841060d0bff
$ wsk action invoke basics/log-beta                                     ❷
ok: invoked /_/basics/log-beta with id 990d284f090c45328d284f090c45320d
$ wsk activation list --limit 2                                         ❸
activations
990d284f090c45328d284f090c45320d log-beta
320b50d841064d0b8b50d841060d0bff log-alpha
$ wsk activation poll --since-seconds 60 --exit 20                      ❹
Enter Ctrl-c to exit.
Polling for activation logs
Activation: 'log-beta' (e6b76a85c5584579b76a85c558957957)
[
    "2018-03-17T17:32:06.364836123Z stdout: beta"
]
Activation: 'log-alpha' (e92e4466ee8f4684ae4466ee8f6684da)
[
```

```
    "2018-03-17T17:32:00.842842699Z stdout: alpha"
]
```

❶ Invoke the action `log-alpha`.

❷ Invoke the action `log-beta`.

❸ Show a list of activations.

❹ Poll the activations (in the last 60 seconds, for 20 seconds) to see which activations happened and what they logged.

Now we are ready to create a trigger, using the command `wsk trigger create`.

Note that triggers are namespace-level entities, and you cannot put them in a package:

```
$ wsk trigger create basics-alert
ok: created trigger alert
$ wsk trigger list
triggers
/openwhisk@example.com_dev/basics-alert                                private
$ wsk trigger get basics-alert
ok: got trigger alert
{
    "namespace": "openwhisk@example.com_dev",
    "name": "basics-alert",
    "version": "0.0.1",
    "limits": {},
    "publish": false
}
```

There are also `update` and `delete` subcommands, and they work as expected, updating and deleting triggers. In the next section we'll see the `fire` subcommand, which requires you to first create rules to do something useful.

Putting the Trigger to Work

Once we have a trigger and some actions, we can create rules for the trigger. A rule is a connection of a trigger with an action, so if you fire the trigger, it will invoke the action. Let's see this in practice. Here, we create a rule, trigger an event, and inspect the logs:

```
$ wsk rule create basics-alert-alpha \
       basics-alert basics/log-alpha                                   ❶
ok: created rule basics-alert-alpha
$ wsk trigger fire basics-alert                                        ❷
ok: triggered /_/alert with id 86b8d33f64b845f8b8d33f64b8f5f887
```

```
$ wsk activation logs 86b8d33f64b845f8b8d33f64b8f5f887 \                    ❸
   | jq                                                                    ❹
{
  "statusCode": 0,
  "success": true,
  "activationId": "b57a1f1dc3414b06ba1f1dc341ab0626",                       ❺
  "rule": "openwhisk@example.com_dev/basics-alert-alpha",
  "action": "openwhisk@example.com_dev/basics/alpha"
}

$ wsk activation logs b57a1f1dc3414b06ba1f1dc341ab0626                      ❻
2018-03-17T18:10:48.471777977Z stdout: alpha
```

❶ Create a rule to activate the action `basics/log-alpha` when the trigger `basics-alert` is fired.

❷ Fire the rule; it returns an activation ID.

❸ Let's inspect the activation ID.

❹ We pipe the output into the `jq` utility to make it more readable.

❺ The rule invoked an action with this activation ID.

❻ Inspect what the rule did.

You can also execute `list`, `update`, and `delete` by name.

A trigger can enable multiple rules, so firing one trigger can actually activate multiple actions. Let's try this feature. But first, let's open another terminal window and enable polling (with the command `wsk activation poll`) to see what happens:

```
$ wsk rule create basics-alert-beta basics-alert basics/log-beta
ok: created rule basics-alert-beta
$ wsk trigger fire basics-alert
ok: triggered /_/basics-alert with id a731a03603bb4183b1a03603bb8183ce
```

If we check the logs we should see something like this:

```
$ wsk activation poll
Enter Ctrl-c to exit.
Polling for activation logs

Activation: 'alert' (a731a03603bb4183b1a03603bb8183ce)                      ❶
[
```

```
"{\"statusCode\":0,\"success\":true,\
\"activationId\":\"3024596c57ac4c10a4596c57ac7c1042\",\
\"rule\":\"openwhisk@example.com_dev/basics-alert-alpha\",\
\"action\":\"openwhisk@example.com_dev/basics/log-alpha\"}",
"{\"statusCode\":0,\"success\":true,\
\"activationId\":\"6d88836c860d405f88836c860d305f83\",\
\"rule\":\"openwhisk@example.com_dev/basics-alert-beta\",\
\"action\":\"openwhisk@example.com_dev/basics/log-beta\"}"
]

Activation: 'log-alpha' (3024596c57ac4c10a4596c57ac7c1042)                    ❷
[
    "2018-03-17T18:34:58.633797676Z stdout: alpha"
]

Activation: 'log-beta' (6d88836c860d405f88836c860d305f83)                     ❸
[
    "2018-03-17T18:34:58.629413468Z stdout: beta"
]
```

❶ The trigger activation invoked two actions.

❷ This is the log of the first action.

❸ This is the log of the second action.

Rules can also be enabled and disabled without removing them. As the last example, let's disable the first rule and fire the trigger again to see what happens. As before, first, we start the log polling to see what happened:

```
$ wsk rule disable basics-alert-alpha                                         ❶
ok: disabled rule basics-alert-alpha
$ wsk trigger fire basics-alert                                               ❷
ok: triggered /_/basics-alert with id 0f4fa69d910f4c738fa69d910f9c73af
```

❶ Disable the rule `alert-alpha`.

❷ Fire the trigger again.

If we check the result, we see that only the action `log-beta` was invoked this time:

```
$ wsk activation poll
Enter Ctrl-c to exit.
Polling for activation logs

Activation: 'basics-alert' (0f4fa69d910f4c738fa69d910f9c73af)
[
    "{\"statusCode\":0,\"success\":true,\"activationId\":\
    \"a8221c7d7fe94e22a21c7d7fe9ce223c\",\
    \"rule\":\"openwhisk@example.com_dev/alert-beta\",\
    \"action\":\"openwhisk@example.com_dev/basics/log-beta\"}",
```

```
    "{\"statusCode\":1,\"success\":false,\
    \"rule\":\"openwhisk@example.com_dev/basics-alert-alpha\",\
    \"error\":\"Rule 'openwhisk@example.com_dev/basics-alert-alpha' is inactive,\
    action 'openwhisk@example.com_dev/basics/log-alpha'\
    was not activated.\",\
    \"action\":\"openwhisk@example.com_dev/basics/log-alpha\"}"
]


Activation: 'log-beta' (a8221c7d7fe94e22a21c7d7fe9ce223c)
[
    "2018-03-18T07:27:14.01530577Z  stdout: beta"
]
```

Using a Feed

Triggers are useful if someone can enable them. You can fire your triggers in code, as we will see when we examine the API.

However, triggers are really there to be invoked by third parties and hook them into our code. This feature is provided by the concept of *feed* (Figure 3-3).

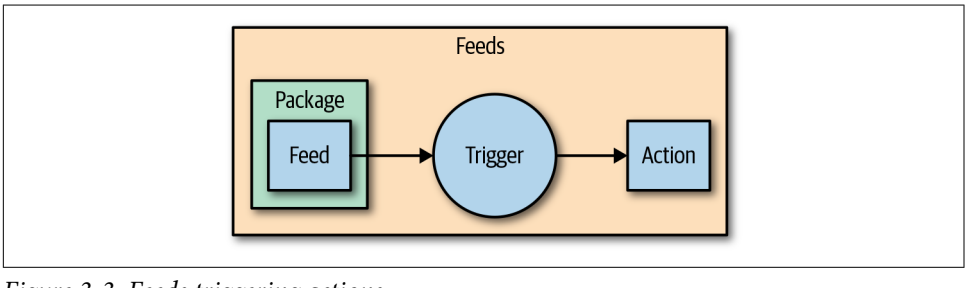


Figure 3-3. Feeds triggering actions

A feed is actually an action that implements a pattern, not an API. We will get into this when we discuss the Observer pattern in the next chapter, but for now let's focus on hooking an existing feed using the command line to create a trigger. For this purpose we will use the `/whisk.system/alarms` package. As we can see, it offers a few actions:

```
$ wsk action list /whisk.system/alarms
actions
/whisk.system/alarms/interval                                  private nodejs:6
/whisk.system/alarms/once                                      private nodejs:6
/whisk.system/alarms/alarm                                     private nodejs:6
```

The `once` feed will trigger an event only once, while `interval` can provide it based on a fixed schedule. The `alarm` trigger is more complex since it uses a `cron`-like expression (not covered here).

Let's create a trigger to be executed every minute using `interval` and associate it with the rule `log-alpha`. As before, we start by polling the logs to see what happens:

```
$ wsk trigger create basics-interval \
  --feed /whisk.system/alarms/interval \                    ❶
  --param minutes 1                                         ❷
ok: invoked /whisk.system/alarms/interval with\
id 5d4bf01d0a56412d8bf01d0a56512d38
{
    "activationId": "5d4bf01d0a56412d8bf01d0a56512d38",
    "annotations": [
        {
            "key": "path",
            "value": "whisk.system/alarms/interval"
        },
        {
            "key": "waitTime",
            "value": 34
        },
        {
            "key": "kind",
            "value": "nodejs:6"
        },
        {
            "key": "limits",
            "value": {
                "logs": 10,
                "memory": 256,
                "timeout": 60000
            }
        },
        {
            "key": "initTime",
            "value": 320
        }
    ],
    "duration": 1153,
    "end": 1521359853176,
    "logs": [],
    "name": "basics-interval",
    "namespace": "openwhisk@example.com_dev",
    "publish": false,
    "response": {
        "result": {
            "status": "success"
        },
        "status": "success",
        "success": true
    },
    "start": 1521359852023,
    "subject": "openwhisk@example.com",
    "version": "0.0.6"
```

```
}
ok: created trigger interval

$ wsk rule create \                                                    ❸
  basics-interval-alpha basics-interval basics/log-alpha
ok: created rule interval-alpha
```

❶ The parameter `--feed` connects the trigger to the feed.

❷ We pass a parameter to the feed, saying we want the trigger to be activated every minute.

❸ The trigger does nothing until we associate it to at least an action with a rule.

If we now wait a couple of minutes, this is what we will see in the activation log:

```
Polling for activation logs:

Activation: 'log-alpha' (0ca4ade11e73498fa4ade11e73a98ff0)
[
    "2018-03-18T08:01:03.046752324Z stdout: alpha"
]

Activation: 'basics-interval' (065c363456b440489c363456b4c04864)
[
    "{\"statusCode\":0,\"success\":true,\
    "activationId\":\"0ca4ade11e73498fa4ade11e73a98ff0\",\
    "rule\":\"openwhisk@example.com_dev/basics-interval-alpha\",\
    "action\":\"openwhisk@example.com_dev/basics/log-alpha\"}"
]
```

After you do this test, do not forget to remove the trigger or it will stay there forever, consuming actions. You may even end up getting a bill for it!

To remove the trigger and the rule:

```
$ wsk rule delete basics-interval-alpha
ok: deleted rule interval-alpha
$ wsk trigger delete basics-interval
ok: deleted trigger interval
```

Generic JavaScript APIs

We'll now explore the JavaScript API, which resembles the CLI.

We do not go into the details of the JavaScript language, as that is out of the scope of this book: there are plenty of sources you can consult to learn about it in depth. This API is available inside the runtime, so you do not need to install it.

However, before discussing the API we need to discuss JavaScript *promises*, as the OpenWhisk API is based on them. Promises are an essential part of asynchronous invocation and are widely used in the OpenWhisk API.

Asynchronous Invocation

A promise is a way to wrap an asynchronous computation in an object that can be returned as a result of a function invocation. To understand why we need to do that, let's consider as an example a simple script that generates a new universally unique identifier (UUID) using the website *httpbin.org*. Here we use the standard Node.js `http` API.

The following code opens a connection to the website and then defines a couple of callbacks to handle the data and error events. In JavaScript, providing callbacks is the standard way of handling events that do not happen immediately; these are the so-called "asynchronous" computations:

```
var http = require("https")

function uuid(callback) {
   http.get("https://httpbin.org/uuid",
   function(res) {
     res.on('data', (data) =>
        callback(undefined, JSON.parse(data).uuid))
   }).on("error", function(err) {
     callback(err)
   })
}
```

Note that this function does not return anything. To retrieve the result and do something with it (e.g., print it on the console), we have to pass a function like this:

```
uuid(function(err, data) {
   if(err) console.log("ERROR:" +err)
   else console.log(data)
})
```

We can use this code locally with Node.js, but we cannot deploy it as an action because it does not return anything. The `main` action must always return "something,"

and in OpenWhisk, the "something" expected for asynchronous computation is a "promise" object. But what is a promise? How do you create one? We cover that next.

Using Promises

We just saw an example of an asynchronous computation. The code in the last listing does not return anything, because the computation is asynchronous. Instead of waiting for the result, we provide a function as a parameter. It will perform our work later, when the computation finally completes.

In OpenWhisk we cannot do this. We need a way to return the asynchronous computation as a result. So, the computation must be wrapped in some object we can return as an answer to the action. The solution for this requirement is called a *promise*.

A promise is an object returned by a function requiring asynchronous computations. Using promises we can take advantage of being asynchronous while transforming the asynchronous computation into an object.

It is essential to understand that even with a promise, the computation is still asynchronous, so the result can be retrieved only when ready. Furthermore, an asynchronous computation can fail with an error.

Promises combine a nice way to retrieve results and error management. With the promise-based action `genuuid` we cover next, extracting the value and managing the errors will be done with the following code:

```
var p = uuid()
p.then(data => console.log(data))
 .catch(err => console.log("ERROR:"+err))
```

Creating a Promise

Let's translate the `genuuid` example to make it promise based. Instead of requiring a callback, we wrap our asynchronous code in a function block like this:

```
return new Promise(function(resolve, reject) {                    ❶
    // OK
    resolve(success)                                              ❷
    // K.O.
    reject(error)                                                 ❸
})
```

❶ Wrapper function to create an isolated environment.

❷ What to do when you get the result successfully.

❸ What to do when you catch an error.

It might look a bit twisted, but it makes sense. A function wraps asynchronous code for three reasons:

- To create an isolated scope
- To be invoked only when the promise resolution is required
- To use the arguments to pass to the code block two functions for returning the result and catching errors

So, the real work of wrapping the asynchronous behavior is performed by the function. But there is more work to do, and we cannot do this work only in the function. We need to:

- Start the invocation when needed (`then`).
- Catch errors (`catch`).
- Provide our implementation of `resolve`.
- Provide our implementation of `reject`.

Hence, we further wrap the function in a promise object and return it as a result. We can now implement our `genuuid` function. Translating the callback is straightforward:

```
var http = require("https")

function uuid() {
  return new Promise(function(resolve, reject){                    ❶
    http.get("https://httpbin.org/uuid",
     function(res) {
      res.on('data', (data) =>
         resolve(JSON.parse(data).uuid))                           ❷
     }).on("error", function(err) {
         reject(err)                                               ❸
    })
  })
}
```

❶ Wrap the asynchronous code in a promise.

❷ Success; we can return the result with `resolve`.

❸ Failure; we return the error with `reject`.

As you can see, the code is similar to the standard way we use a callback, except we wrap everything in a function that provides the `resolve` and `reject` actions. The function is then used to build a promise. Most of the implementation follows the standard way; only a function must be defined to implement a specific behavior.

Now, using the promise, we can create an action for OpenWhisk, providing the following `main`:

```
function main() {                                                    ❶
    return uuid()
      .then( data =>  ({"uuid": data}))
      .catch(err =>   ({"error": err}))
}
```

❶ We do not specify `args` here because we do not use them.

Let's try the code:

```
$ wsk action update apidemo/promise-genuuid \
      apidemo/promise-genuuid.js
ok: updated action apidemo/promise-genuuid
$ wsk action invoke apidemo/promise-genuuid -r
{
    "uuid": "52a188a9-ff9b-4f3a-b72d-301c26aac2cf"
}
```

Using the OpenWhisk API

To access the OpenWhisk API you use the standard Node.js `require` commmand. Every action that needs to interact with the system must start with:

```
var openwhisk = require("openwhisk")
```

However, the `openwhisk` object is just a constructor used to access the API. You need to construct the actual instances to access the methods of the OpenWhisk API:

```
var ow = openwhisk()
```

Hidden in this call there is the actual work of connecting to the API server and dialoguing with it. This call is a short form that uses implicitly some parameters to perform a connection; the long form is as follows:

```
var ow = openwhisk({
 apihost: <your-api-server-host>,
 api_key: <your-api-key>
})
```

If you use the shortened form, `apihost` and `apikey` are retrieved from two environment variables:

- `__OW_API_HOST`
- `__OW_API_KEY`

When an action is running inside OpenWhisk, these environment variables are already set for you, so you can use the initialization without passing them explicitly.

While you usually `require` the API in the body of your action, outside of any function, you need to perform this second step *inside* the `main` function or any function called by it. This is a known issue in OpenWhisk: environment variables are not yet set when you load the code. They are set only when you invoke `main`. Hence, if you perform the initialization outside of `main`, it will fail.

If you want to connect to OpenWhisk from the outside (e.g., from a client application), you can still use the OpenWhisk API, but you have to either set the environment variables or pass the API host and key explicitly.

If you are using, for example, the IBM Cloud, you can retrieve your current values with `wsk property get --auth`.

The OpenWhisk API provides the same functions that are available in the CLI. Some features are more useful than others for actually writing action code. The most important for our purposes are `ow.actions`, `ow.triggers`, and `ow.activations`. We'll look at those next.

The API also has features that are useful for deploying code. We will not discuss those features here, since they are needed only when you develop your deployment tools. Those other available features are `ow.rules`, `ow.packages`, and `ow.feeds`. You can find detailed coverage of them on GitHub, should you need them.

Invoking OpenWhisk Actions

Probably the most critical API function available is `invoke`. It is used from an action to execute other actions, so it is essential to build interactions between various actions and it is the backbone of complex applications.

In the simplest case you just use the following:

```
var ow = openwhisk()
var pr = ow.actions.invoke("action")
```

Note that by default the action invocation is asynchronous. It just returns an activation ID, so if you need the results, you have to retrieve them by yourself. If instead you have to wait for the execution to complete, you add `blocking: true`. If you also need the results, and not just the activation ID, you add `result: true`. For example, you can invoke the `promise-genuuid` action we implemented before from a web action, so that a UUID can be shown from a web page, as follows :

```
var openwhisk = require("openwhisk")                                    ❶
function main(args) {
    let ow = openwhisk()                                                ❷
    return ow.actions.invoke({                                          ❸
        name: "apidemo/promise-genuuid",
        result: true,
        blocking: true
    }).then(res => ({                                                   ❹
         body: "<h1>UUID</h1>\n<pre>"+
        res.uuid+"</pre>"
    })).catch(err => {statusCode: 500})
}
```

❶ Require the API.

❷ Instantiate the API access object.

❸ Actual action invocation (note we want to immediately use the result).

❹ Get the value returned by the other action.

The web action invokes the other action, waits for the result, extracts it, and builds a simple HTML page. For example:

```
$ wsk action update apidemo/invoke-genuuid \                            ❶
  apidemo/invoke/invoke-genuuid.js  --web true
ok: updated action apidemo/invoke-genuuid
$ curl $(wsk action get apidemo/invoke-genuuid --url \                  ❷
  | tail -1)
<h1>UUID</h1>                                                           ❸
<pre>d9333213-be20-449d-adc9-ecd8224772ad</pre>
```

❶ Deploy the action as a web action.

❷ Extract the URL of the action and then use `curl` to invoke the URL.

❸ The generated UUID, presented as HTML markup.

Accessing OpenWhisk from the outside

In this example, we built the `ow` object without parameters. This object requires authentication parameters, but if you do not pass them it gets them from the environment variables. In this way, an action can invoke another action that belongs to the same users. However, it is also possible to use this API to invoke actions that belong to other users, or from Node.js applications running outside of OpenWhisk. In those cases, you have to provide the authentication information explicitly. The general format of the API constructor is as follows:

```
let ow = openwhisk({
   apihost: host,
   api_key: key
}
```

In the cases in which you are running from outside an action, you have to get `api host` and `api_key` and use them in the API constructor to be able to invoke other actions or triggers. That information can be either retrieved using the `wsk property get` command or read from the environment variables `__OW_API_HOST` and `__OW_API_KEY` with an action.

Invoking multiple promises

We know how to use promises to chain asynchronous operations, but what about when we have multiple promises?

We could wait for multiple actions with some complex code involving `then`. But since this use case is frequent enough, it merited the availability of a standard method: `Promise.all(promises)`.

Within this method call, `promises` is an array of promises (actually, anything satisfying the "iterable" JavaScript protocol). As a result, `Promise.all()` returns a promise that completes when all the other promises are complete. You can then retrieve the combined result of all the promises with a `then` function, receiving, as a result, an array of the results of the many promises.

Let's demonstrate this feature with an example. We are going to create a web action consisting of multiple promises (actually, multiple invocations). We will wait for all of them to complete, and then we will construct a web page concatenating the results. You may notice in the example the use of `map`, which is probably idiomatic in these cases:

```
var openwhisk = require("openwhisk")

function main(args) {
    let ow = openwhisk()
    let count = args.n ? args.n : 3;                                    ❶
    let inv = { name: "apidemo/promise-genuuid",
                blocking: true,
                result: true
    }
    let promises = []
    for(let i=0; i< count; i++)                                         ❷
        promises.push(ow.actions.invoke(inv));
    return Promise.all(promises)                                        ❸
      .then(res => ({
            body: "<h1>UUIDs</h1><ul><li>"+
                   res.map(x=>x.uuid).join("</li><li>")+                ❹
                   "</li></ul>"
```

```
        }))
}
```

❶ Read a parameter from the URL; if not specified it defaults to `3`.

❷ Loop and construct an array of invocations, each one asking for a different UUID.

❸ This is the key call: we are waiting for all the promises to complete.

❹ The final result is built here, in the form of an array of results, each one being a different UUID. We join them to create a list in HTML.

Now, we can test the invocation and check the results:

```
$ wsk action update apidemo/invoke-genuuid-list \                    ❶
  apidemo/invoke/invoke-genuuid-list.js  --web true
ok: updated action apidemo/invoke-genuuid-list
$ curl $(wsk action get apidemo/invoke-genuuid-list  \               ❷
         --url | tail -1)
<h1>UUIDs</h1>                                                       ❸
<ul>
  <li>3b72aec9-eb1d-45e9-9c98-d689a6bddf2b</li>
  <li>3189605c-d003-469b-afcc-202ef690a544</li>
  <li>3e78ee6d-f141-483b-8d1a-7e8aaaae309d</li>
</ul>
```

❶ Deploy the action wrapping multiple promises as a web action.

❷ Invoke of the new action using `curl` and extract the URL.

❸ The result is now the combination of multiple invocations of the `promise-genuuid` action.

Firing Triggers

Now we are going to see how to fire triggers with the API. This is especially useful for implementing feeds, as we'll see in "Observer" on page 110. It works pretty much like action invocation, except you have to use `triggers` instead of `actions`. Also, firing a trigger cannot be blocking, because it can always start multiple actions. Instead, it always returns a list of activation IDs.

Keep in mind that you may need to fire triggers belonging to other users. Generally, you fire triggers as part of the implementation of a feed. We cover that in Chapter 5. When you create a trigger and hook it into a feed, you receive from the system, among other parameters, an `api-key`. You have to use this key to invoke those external triggers.

Here is an example of a proper invocation of a third-party trigger:

```
const openwhisk = require('openwhisk')

function fire(trigger, key, args) {
  let ow = openwhisk({ api_key: key })                    ❶
  return ow.triggers.invoke({                             ❷
      name: trigger,                                      ❸
      params: args                                        ❹
  })
}
```

❶ Construct an instance of the API, passing the `api_key`.

❷ Perform the invocation.

❸ Specify the trigger name.

❹ Some of the parameters that will be forwarded to the triggered actions.

The result of an invocation is a promise that returns a JSON object like this:

```
{
    "activationId": "a75b8008ad2641189b8008ad26f11835"
}
```

You can then use the activation ID to retrieve the results, as explained shortly.

Now let's try to deploy and run this code. For this example, we'll reuse the *clock.js* code from "Including Some Code of Your Own as a Library" on page 55. We need to set up an action activated by a trigger to test it:

```
$ wsk trigger create apidemo-trigger                                    ❶
ok: created trigger apidemo-trigger
$ wsk action update apidemo/clock apidemo/trigger/clock.js              ❷
ok: updated action apidemo/clock
$ wsk rule update apidemo-trigger-clock apidemo-trigger apidemo/clock   ❸
ok: updated rule apidemo-trigger-clock
$ wsk action update  apidemo/trigger apidemo/trigger/trigger.js         ❹
ok: updated action apidemo/trigger
```

❶ Create a trigger.

❷ Create an action the trigger will invoke.

❸ Create a rule that connects the trigger to the action.

❹ Finally, deploy the action dependent on the trigger.

Now everything is set up: invoking the action `apidemo/trigger` will activate the action `apidemo/clock` via a trigger and a rule. You can test it by opening polling with `wsk activation poll` and then doing:

```
$ wsk action invoke apidemo/trigger -p time 1
ok: invoked /sciabarra_cloud/apidemo/trigger with\
id 1c453cc7142c4e19853cc7142c3e1998
$ wsk action invoke apidemo/trigger -p date 1
ok: invoked /sciabarra_cloud/apidemo/trigger with\
id 3d157c76dfa244ce957c76dfa294cea3
```

In the activation log, you should see (among other things) the logs emitted by the `clock` action that was activated by the trigger:

```
Activation: 'clock' (bdc6438e11b34bb686438e11b3cbb67c)
[
    "2018-09-25T17:13:24.584124671Z stdout: 17:13:24"
]
Activation: 'clock' (4fc0889664ad4e1880889664ad5e18e0)
[
    "2018-09-25T17:13:40.667002422Z stdout: 2018/09/25"
]
```

Inspecting Activations

Now, let's use the API to inspect activations. An activation ID is returned by invoking an action or firing a trigger. Once you have this ID, you can retrieve the entire record using `ow.activations.get`, as in the following example (*activation.js*):

```
const openwhisk = require('openwhisk')

function main (args) {
  let ow = openwhisk()
  return ow.activations.get({ name: args.id })
}
```

As you can see, you need to pass the ID (as the parameter named `name` by convention, but the parameter `id` also works) to retrieve results, logs, or the entire record activation. Let's try it, first deploying the example and then creating an activation ID:

```
$ wsk action create activation activation.js                    ❶
ok: created action activation
$ wsk action invoke \
   /whisk.system/samples/helloWorld \                           ❷
  -p payload Mike
```

```
ok: invoked /whisk.system/samples/helloWorld with \
id a6d34e7165024c26934e716502fc26cf
```

❶ Deploy the `activation` action.

❷ Execute a system action to produce the activation ID.

Now we can use that ID to retrieve the activation, which is a pretty complex (and interesting) data structure. Here are the results (shortened for clarity):

```
$ wsk action invoke activation -p id a6d34e7165024c26934e716502fc26cf -r
{
    "activationId": "a6d34e7165024c26934e716502fc26cf",
    "annotations": [
       // annotations removed...
     ],
    "duration": 4,
    "end": 1522237060018,
    "logs": [                                                          ❶
        "2018-03-28T11:37:40.016763747Z stdout: hello Mike!"
    ],
    "name": "helloWorld",
    "namespace": "openwhisk@example.com_dev",
    "publish": false,
    "response": {
        "result": {},                                                  ❷
        "status": "success",
        "success": true
    },
    "start": 1522237060014,
    "subject": "openwhisk@example.com",
    "version": "0.0.62"
}
```

❶ The logs of the action.

❷ The result of the action.

We used `ow.activations.get` in this example to get the entire activation record. You can use the same parameters but specify `ow.0.result` to get just the results, or `ow.activations.logs` to retrieve just the logs.

Summary

OpenWhisk has in practice one access point: its REST API. This API is used in two ways: interactively, by using a CLI, and programmatically, by writing code. In this chapter, we explored both ways to use the API.

First, we covered the various commands, subcommands, and options of the CLI. Then we learned how to create actions, sequences, and triggers, deploying either single- or multiple-file actions. We also discussed how to retrieve logs and results. Finally, we learned about the JavaScript API. We covered promises, an important part of this API, and then got into action invocation and firing triggers.

CHAPTER 4

Common Design Patterns in OpenWhisk

This chapter and the next focus on designing OpenWhisk applications. When writing your code, you need to have a good grasp of programming languages and algorithms. In this chapter, we'll apply many of the classic "Gang of Four" design patterns as described in *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley). We also cover the ubiquitous Model-View-Controller pattern, although it is not part of this classic collection. We will go over an example for each pattern, and in the process, revamp our contact form from Chapter 2. To illustrate how to build and engineer a real serverless application using a design pattern, we will create a contact form handler.

Figure 4-1 shows the patterns covered in this chapter: Singleton, Facade, Prototype, Strategy, Chain of Responsibility, and Command. We'll cover a few more patterns in Chapter 5.

The source code for the examples in this chapter and the next are available in the book's GitHub repository.

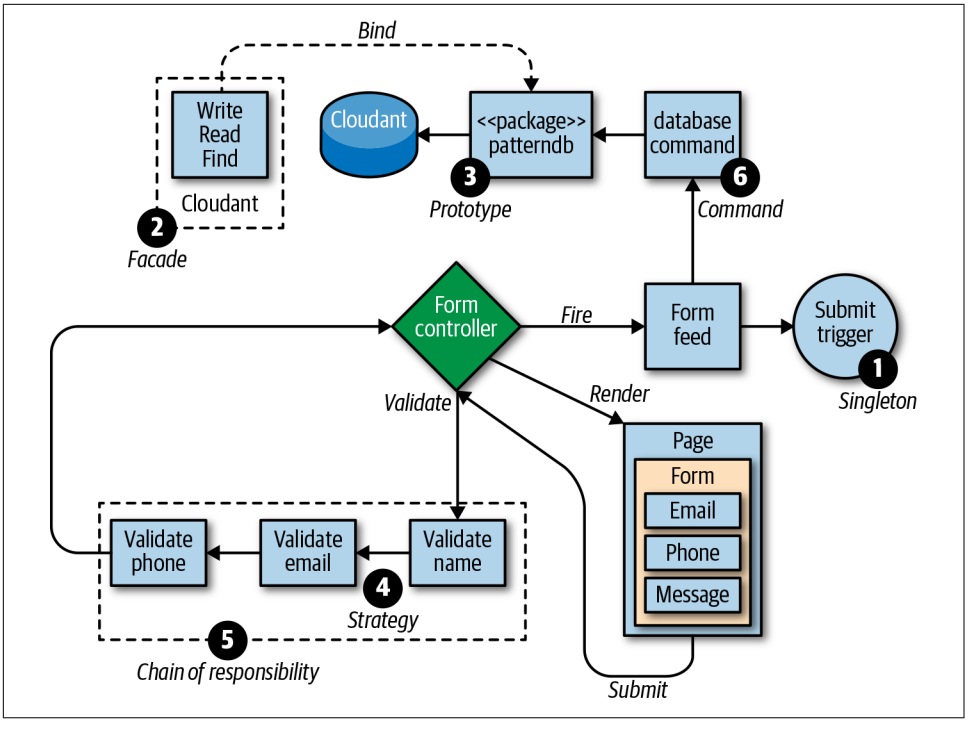


Figure 4-1. A contact form using common design patterns

Built-in Patterns

OpenWhisk allows you to combine actions that implement some of the more common patterns out of the box. These patterns include:

Singleton
: Available through triggers and named invocations (**1** in Figure 4-1)

Facade
: Provided by packages with web actions and feeds (**2** in Figure 4-1)

Prototype
: Used when you bind packages (**3** in Figure 4-1)

Decorator
: Available as the annotation feature (not shown in the diagram)

Singleton

The Singleton pattern (Figure 4-2) meets the challenge of locating services and components in an application with many parts, to coordinate the work across the different parts of the application. This is probably the most straightforward pattern. A singleton restricts the number of instances of a class to a single instance, and provides easily accessible methods to reach that instance, using some global name (see Figure 4-1).

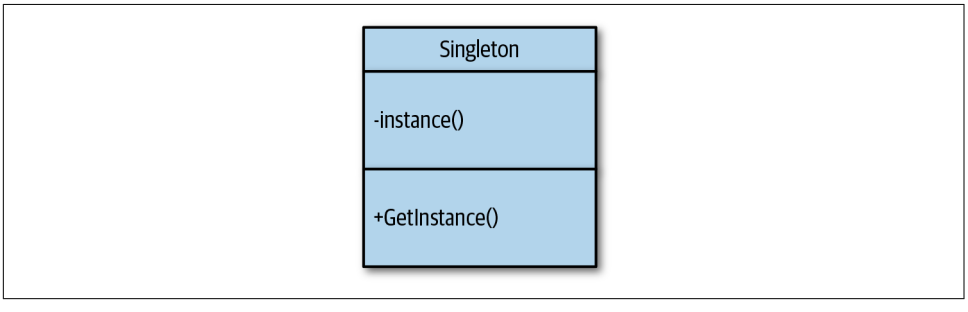


Figure 4-2. The Singleton pattern

In general, you can implement this pattern using static methods or global variables to retrieve the unique instance of a class. Another option is to use libraries that do dependency injection and manage named instances of objects.

In OpenWhisk, the Singleton pattern is embedded in the API so you can retrieve actions by name. A trigger is a type of Singleton pattern, because it is used to locate a single point of access to multiple actions.

In a sense, the fact that actions are named and can be invoked using their name as an implementation of the Singleton pattern. You can consider each action as the unique instance of an (implicit) class containing `main` as a method.

In our contact form, we use a trigger as a unique entry point for processing the form submission (Figure 4-3). The trigger `submit` is in charge of receiving data coming from the frontend. It acts as a Singleton since the form processes the submission by name. What happens next depends on the configuration of the application. For example, as we are going to see, we can create rules to process a form submission, store it in the database, and send an email.

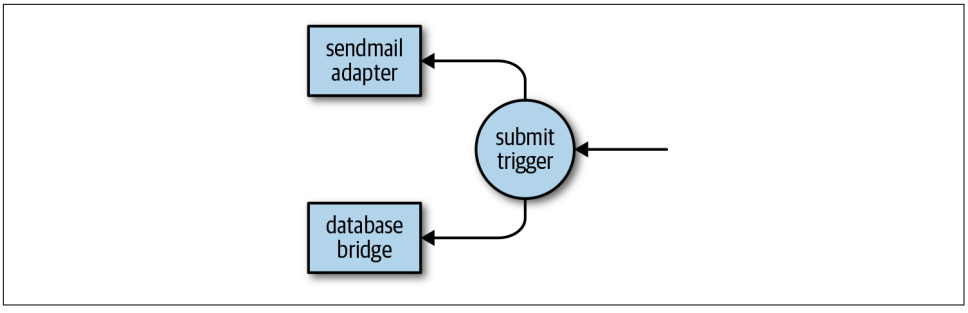


Figure 4-3. The Singleton pattern used to implement a trigger

Another valid view of the pattern implemented by OpenWhisk when instantiating actions is the AbstractFactory pattern: there is indeed a single interface that instantiates different implementations of the actions as provided by runtimes.

Facade

The Facade pattern makes a complex subsystem easier to use by providing a more straightforward interface to it (see Figure 4-4). You use the Facade pattern when you have an application with many subsystems and you want to simplify their use. A facade usually provides only the subset of the functionalities of the subsystems that are of interest to users.

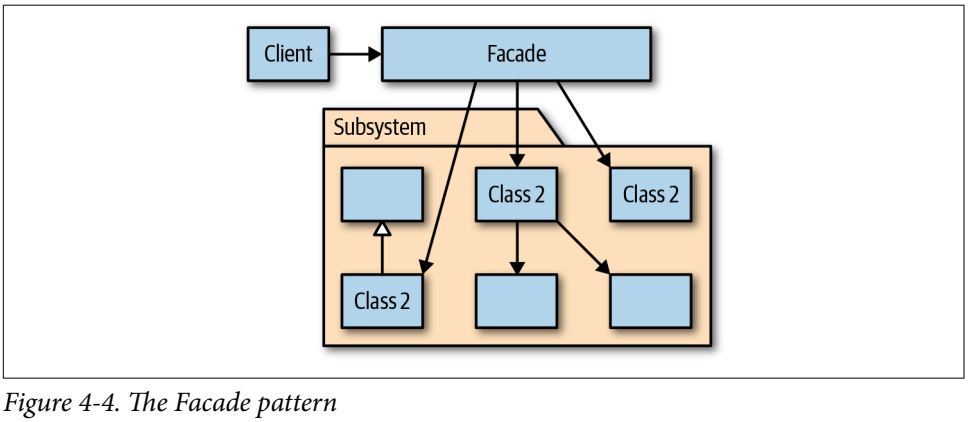


Figure 4-4. The Facade pattern

In OpenWhisk we can design an application using many interconnected actions. Those actions are, by themselves, hiding some complexity and providing a more straightforward interface to complex systems in the cloud, like GitHub or Slack. Thus, those actions could also be considered implementations of the Facade pattern.

OpenWhisk offers a few ways to define an external interface. You can mark actions as web actions, you can expose some actions as feeds, and you can define some actions as triggers (Figure 4-5).

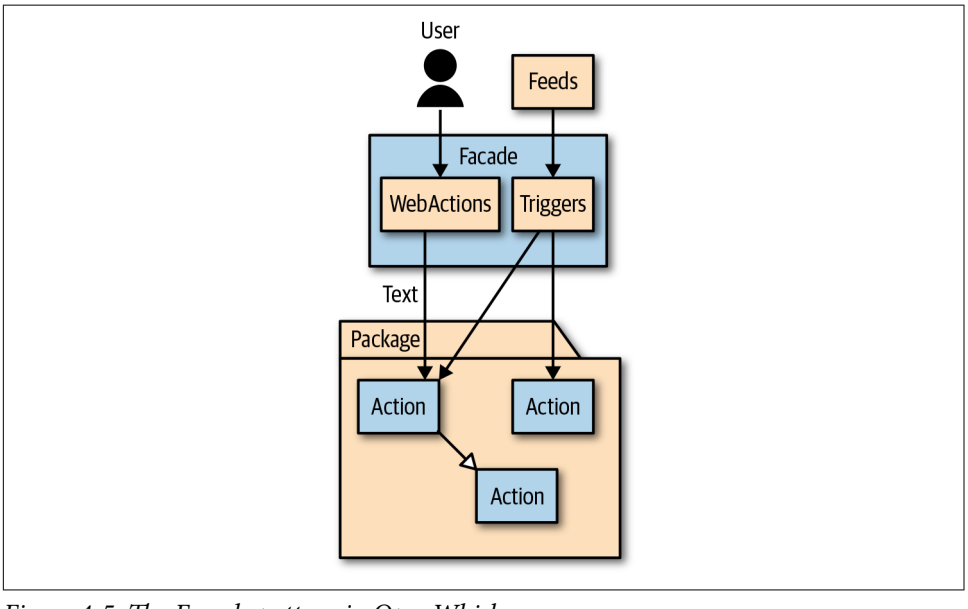


Figure 4-5. The Facade pattern in OpenWhisk

Web actions define the external interfaces to be used by web clients. You can use them to define an externally accessible REST API. Defining a similar API is equivalent to providing the simplified access to the complex processing performed by other actions, as the Facade pattern requires.

But the interface of your Facade does not have to be web-based. You can use triggers as entry points to your application, and you can invoke them without having to expose them publicly. To use them you have to use one of the available client libraries to access the APIs of OpenWhisk. Note, however, that an essential feature of triggers is that you can connect them to feeds, so triggers offer an interface to other OpenWhisk applications.

A Feed is an action that must follow a documented pattern (we'll see an example later in this chapter, when we discuss the Observer pattern). Using Feeds you can connect an OpenWhisk application to another web application, thus providing an interface to use it as a subsystem. While web actions expose a web interface, feeds invoke third-party web interfaces.

Apache OpenWhisk includes many packages that can be considered facades for complex subsystems. The ones that are available out of the box in the IBM Cloud platform include:

- `/whisk.system/github`, a facade over the GitHub API
- `/whisk.system/slack`, a facade to access the Slack API
- `/whisk.system/weather`, which allows access to the service giving information on the weather
- `/whisk.system/watson-translator`, an IBM service for language translation

Prototype

The Prototype pattern makes constructing complex objects easier. It works by creating an instance to be used as a model, or "prototype," for new instances. The creation of new instances of the model is performed using a `clone` method. Of course, generally, it is not that useful to have completely identical copies of a given prototype. It is therefore common to provide parameters to the `clone` method in order to create "customized" clones that are more fit for our purposes (see Figure 4-6).

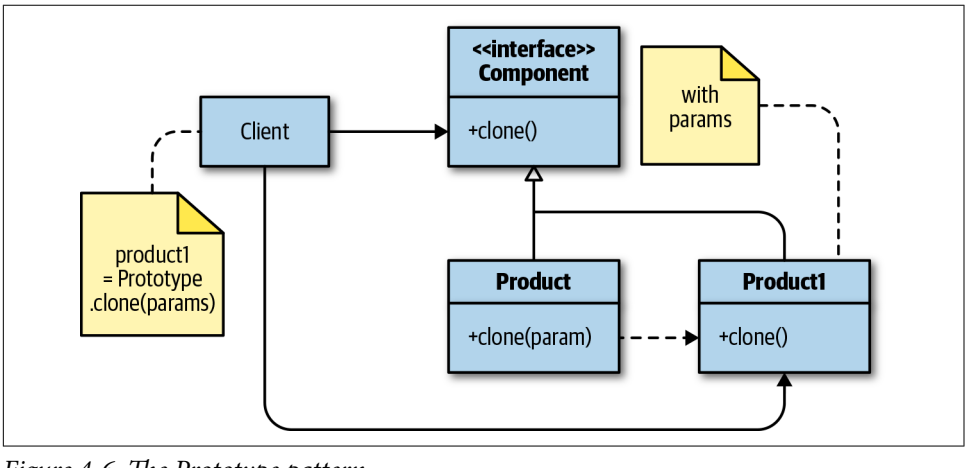


Figure 4-6. The Prototype pattern

The Prototype pattern is used in OpenWhisk in package binding. Using packages, we can create collections of actions that can be shared and reused. Using the command `wsk package bind`, we can create a copy of a package, just like the Prototype pattern prescribes. All the actions are then available under the new package name.

A complete copy of a package is rarely useful. When binding a package, we can provide a set of parameters to customize the package for our needs.

For example, to use Cloudant in IBM cloud, you use a package binding after you provision your server instance. A server instance includes multiple databases. You can create a clone that can read and write in multiple databases by providing username and password. Or better still, you can create a clone that can read and write in a single database within the server instance by providing a username, password, and database name.

We use the Prototype pattern in our contact form, customizing it with a set of parameters. We download a JSON file that includes the parameters to access the server instance, and then use those at bind time to create a cloned package customized with some parameters, so we get a prototype package parametrized to read and write in only one database (see Figure 4-7).

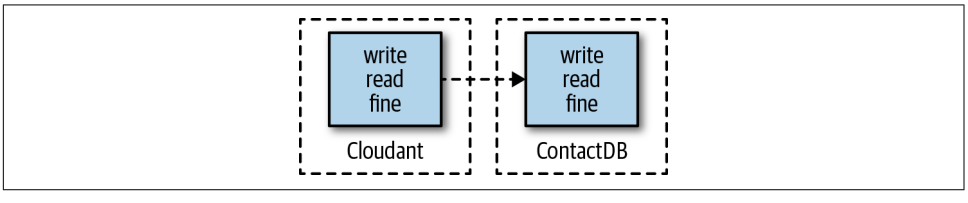


Figure 4-7. The Prototype pattern creating an instance of the Cloudant package

A binding creates a copy of an existing package, but lets you change the parameters. This way you can create an instance of a package customized for your use. We already covered how to bind to a package using specific parameters—now let's see how to bind to a database using a JSON file. This feature implements the Prototype pattern with a custom parameter.

Go into the Cloudant database, as shown in Figure 4-8, and copy the configuration file. Then open a text editor, paste the the JSON file, and save it as *cloudant.json*.

Using that file can you now bind the `patterndb` database as follows:

```
$ wsk package bind /whisk.system/cloudant patterndb \
   -P cloudant.json \                                    ❶
   -p dbname pattern                                     ❷
ok: created binding patterndb
```

❶ The configuration file binds to a server instance.

❷ We add a parameter to use a specific database.

Now you can access all the actions configured in the `cloudant` package. Since we provided a clone customized with our parameters, the actions actually use our database in the server instance.

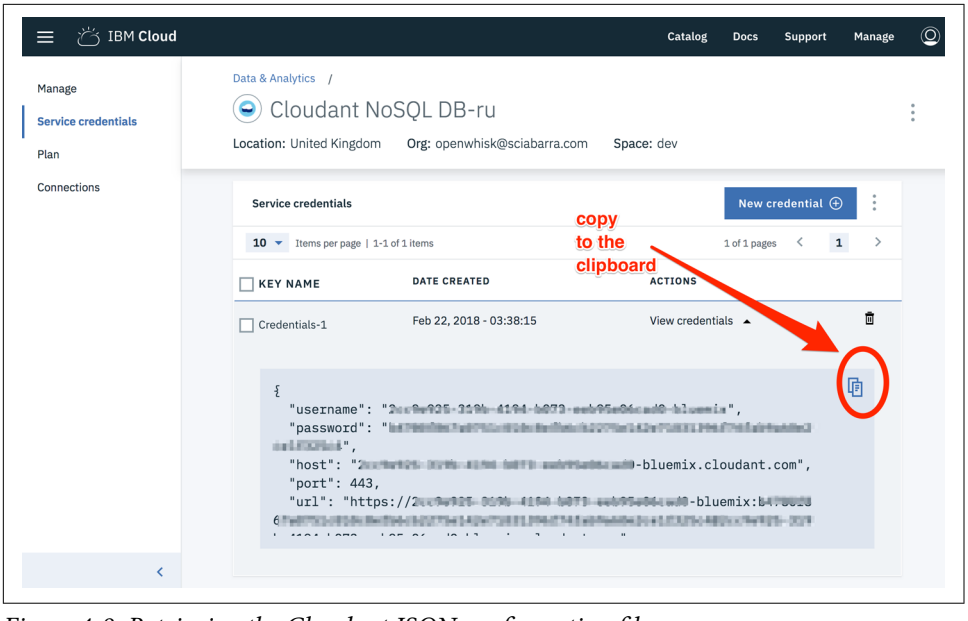


Figure 4-8. Retrieving the Cloudant JSON configuration file

Decorator

The Decorator pattern adds functionalities to an existing system, without changing the interface of the system and while preserving all the old functionality. With this pattern, we take an instance of a class and wrap it with another instance of the same class, providing the same interface. The added functionality is performed by the "wrapper," which will also use existing functionality to perform its work (see Figure 4-9).

In OpenWhisk, the Decorator pattern is implemented using *annotations*. Annotations are a way to add additional information to OpenWhisk entities, and to add tools, plugins, and custom implementations.

Most of the available annotations are for documentation only and change the behavior of an entity only when displaying the online help. However, some annotations change the behavior of an entity in a more "semantic" way.

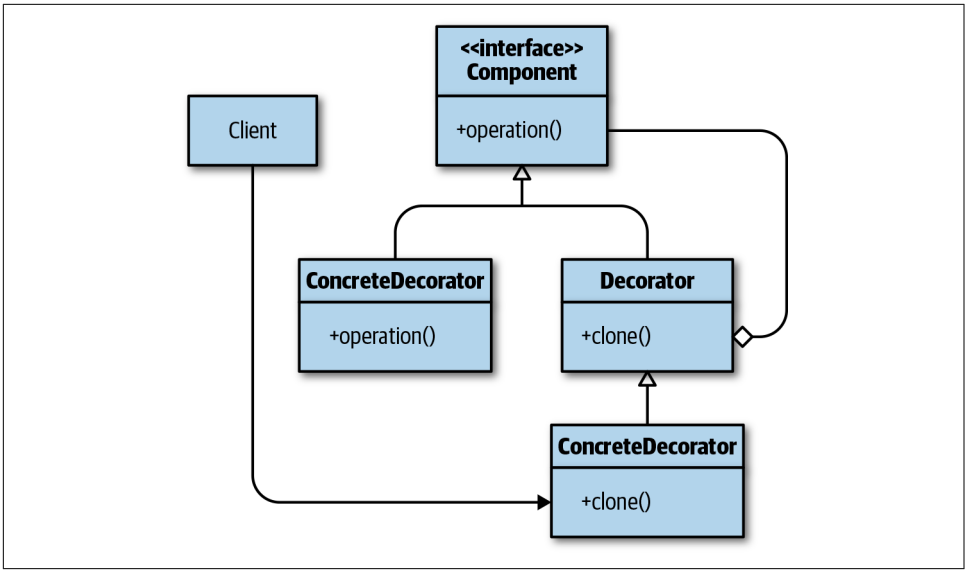


Figure 4-9. The Decorator pattern

In general, you can add an annotation with the flag `-a <name> <value>` for all OpenWhisk entities. Furthermore, you can read the annotation of every entity using the `get` subcommand.

We can use the following annotations to add additional information to our entities:

- parameters for packages and actions show fields:
  - `doclink` (link to documentation)
  - `required` (is required)
  - `bindTime` (was this parameter bound?)
  - `type` (for example `password` or `array`)
- `description` for packages, actions, and parameters
- `sampleInput` and `sampleOutput` for actions

Currently, these annotations are not checked, so it is up to the developer to respect (or ignore) them. At some point in time, however, they may become constraints.

As an example, let's read the annotations to see what parameters are available for the `cloudant` package's `read` action:

```
$ wsk action get /whisk.system/cloudant/read  \
   | tail +2 \                                                    ❶
   | jq .annotations                                             ❷
[
  {
    "key": "description",
    "value": "Read document from database"
  },
  {
    "key": "parameters",
    "value": [
      {
        "name": "dbname",
        "required": true
      },
      {
        "description": "The Cloudant document id to fetch",
        "name": "id",
        "required": true
      },
      {
        "name": "params",
        "required": false
      }
    ]
  },
  {
    "key": "exec",
    "value": "nodejs:6"
  }
]
```

❶ Get rid of the first line to feed JSON to `jq`.

❷ Extract only the `annotations` part of the JSON output.

Now let's see how we can add annotations, for example, to document the `apidemo/clock` interface:

```
$ wsk action update apidemo/clock ok: updated action apidemo/clock
$ wsk action update apidemo/clock \
-a "description" "return the current date or \
time" -a "parameters" '[{"name":"time","description":"show time"}, \
{"name":"time","description":"show time"}']
ok: updated action apidemo/clock
```

Let's check the results:

```
$ wsk action get apidemo/clock  | tail +2 | jq .annotations
[
  {
    "key": "description",
    "value": "return the current date or time"
  },
  {
    "key": "parameters",
    "value": [
      {
        "description": "show time",
        "name": "time"
      },
      {
        "description": "show time",
        "name": "time"
      }
    ]
  },
  {
    "key": "exec",
    "value": "nodejs:6"
  }
]
```

Note the `exec` annotation added automatically by the system.

Useful annotations are also added to activations. We can see them here:

```
$ wsk action invoke apidemo/clock
ok: invoked /_/apidemo/clock with id 17cb42c9fd2a45ad8b42c9fd2a45ad71
$ wsk activation get 17cb42c9fd2a45ad8b42c9fd2a45ad71 \
  | tail +2 | jq .annotations
[
  {
    "key": "path",
    "value": "openwhisk@example.com_dev/apidemo/clock"
  },
  {
    "key": "waitTime",
    "value": 45
  },
  {
    "key": "kind",
    "value": "nodejs:6"
  },
  {
    "key": "limits",
    "value": {
      "logs": 10,
      "memory": 256,
      "timeout": 60000
    }
```

```
    },
    {
      "key": "initTime",
      "value": 74
    }
  ]
```

Some useful information is provided by these annotations:

- `path` is the action associated with this invocation.
- `limits` are the execution constraints of the action.
- `kind` is the runtime used.
- `waitTime` is the time spent before the action is activated.
- `initTime` is the time needed to initialize the action.

In addition to these documentation annotations, there are annotations for web actions, which will be discussed in "Advanced Web Actions" on page 239.

You can also use sequences to implement the Decorator pattern. Consider an action that requires an authentication token to perform its work that must be retrieved automatically. You can "decorate" the token by using a sequence: the first action retrieves the authentication token and passes it to the second, which does its work.

Patterns Commonly Implemented with Actions

There are some other patterns that are used very frequently in the context of OpenWhisk and serverless applications. Hence, we gather them under the hat of "common" patterns. They are not part of OpenWhisk, but they are easily implemented with OpenWhisk actions

We'll cover the following patterns in this section:

Strategy
: Used to change implementations while keeping a standard interface (#4 in Figure 4-1)

Chain of Responsibility
: Used to partition the solution of a problem into multiple steps (#5 in Figure 4-1)

Command
: Used to encapsulate information needed to perform a task (#6 in Figure 4-1)

Strategy

The Strategy pattern provides different functionalities to a system while keeping a uniform interface. It works by defining a base class for an algorithm that acts as the interface. We can then extend this base class with other classes to customize the logic as needed. The client sees a standard interface, while the behavior is implemented by providing a different implementation (or "strategy") for each subclass (see Figure 4-10).

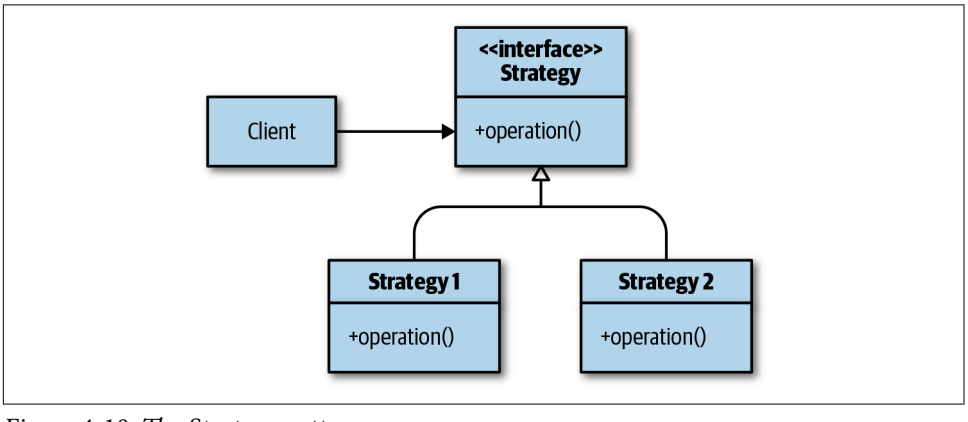


Figure 4-10. The Strategy pattern

To illustrate the Strategy pattern, we will implement validation in our contact form. We will have a base class performing the validation, with a standard interface. Then we will customize the validation logic to behave in a different way when we validate an email address or a phone number, as illustrated in Figure 4-11.

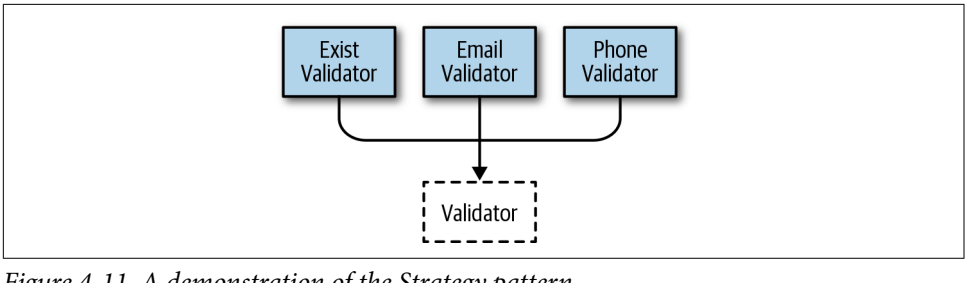


Figure 4-11. A demonstration of the Strategy pattern

To validate the different fields in the form, we'll create a library file *lib/validator.js* with the following contents. The interface is the same for each field ("validate"), but the implementation is different: an email address must be validated in a different way than a phone number. The `Validator` class looks like this:

```
class Validator {
  constructor(field) {
```

```
    this.field = field;
  }

  // simple validator - just check it not empty
  // return an error, or an empty string if ok
  validator(value) {                                          ❶
    if(value)
        return ""
    return "missing "+this.field;
  }

  // validate data, adding messages and values
  validate(data) {                                            ❷
    if (!data.message) data.message = [];
    if (!data.errors)  data.errors = [];

    let value = data[this.field];
    let err = this.validator(value);                          ❸
    if (err) data.errors.push(err);
    else data.message.push(this.field + ": " + value);
    return data;
  }
}
```

❶ The replaceable validation logic; by default, it checks only if a field exists.

❷ The validator gets some data, then applies the validation logic to the specified field.

❸ Here is where we apply the validation logic.

The validator can be used as it is to validate the existence of a single field, as shown here in *name.js*:

```
const Validator = require("./lib/validator.js")               ❶

function main(args) {
    return new Validator("name").validate(args)               ❷
}
```

❶ Import the `Validator` class from the library.

❷ Create a validator for the `name` field, returning the form data annotated with messages and errors.

Now let's apply the Strategy pattern by providing different implementations of the validation logic. First we'll create an `email` action (*email.js*) to validate the email field with a validator for email addresses:

```
const Validator = require("./lib/validator.js")
```

```
class EmailValidator extends Validator {                          ❶
    validator(value) {
        let error = super.validator(value);
        if (error) return error;
        var re = /\S+@\S+\.\S+/;                                  ❷
        if(re.test(value))
            return "";
        return value+" does not look like an email"
    }
}

function main(args) {                                             ❸
    return new EmailValidator("email").validate(args)
}
```

❶ Redefine the `validator` method, hence applying the Strategy pattern.

❷ Validation logic, matching the email field against this regular expression.

❸ We validate in the same way as before, but now we use the `EmailValidator`.

Next we'll use a different strategy, implementing a `phone` action to validate a phone number:

```
const Validator = require("./lib/validator.js");

class PhoneValidator extends Validator {
  validator(value) {
    let error = super.validator(value);
    if (error) return error;
    var match = value.toString().match(/\d/g)                     ❶
    if (match && match.length >= 10) return "";
    return value + " does not look like a phone number";
  }
}

function main(args) {                                             ❷
  return new PhoneValidator("phone").validate(args);
}
```

❶ This validator extracts the numbers from the string, then verifies that there are at least 10 of them.

❷ We can use this validator in the same way as before.

This validation was designed to be chained, which means we can use it to implement another pattern—Chain of Responsibility.

Chain of Responsibility

The Chain of Responsibility pattern splits large, complex processing into multiple smaller, separate steps that are assembled as a chain of execution.

It works by doing the following (see Figure 4-12):

- Defining a standard interface for the task to be performed
- Implementing each step of the processing with a separate class
- Connecting the specific instances performing the processing in a chain—when one step is complete, the next is called to continue the processing

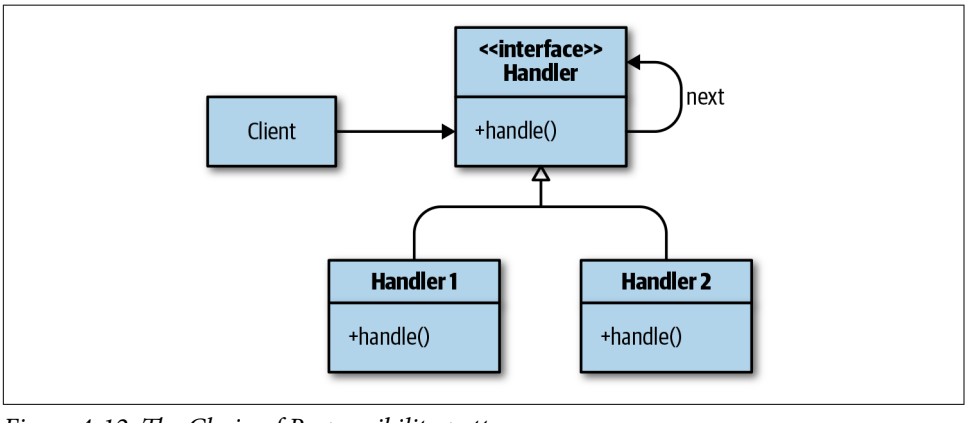


Figure 4-12. The Chain of Responsibility pattern

The Chain of Responsibility pattern in OpenWhisk can be implemented in a natural way by using sequences. Of course, to put actions in a sequence, the output of an action must be usable as the input of the next action in the sequence (Figure 4-13).

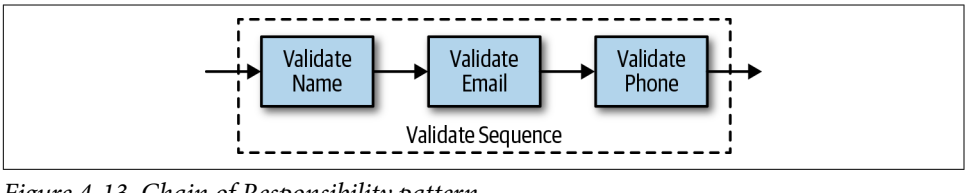


Figure 4-13. Chain of Responsibility pattern

Actually, the implementation of the Strategy pattern in the preceding section is also an implementation of the Chain of Responsibility pattern.

Let's review the `Validator` class again, to see where the pattern is. The key concepts are:

- We need to partition the problem to apply different instances of the chain.

- We have to make sure the instances can be concatenated.

Here's the *lib/validator.js* file again:

```
class Validator {
  constructor(field) {                                          ❶
    this.field = field;
  }

  // simple validator - just check if not empty
  // return an error, or an empty string if OK
  validator(value) {
    if(value)
        return ""
    return "missing "+this.field;
  }

  // validate data, adding messages and values
  validate(data) {
    if (!data.message) data.message = [];                       ❷
    if (!data.errors)  data.errors = [];

    let value = data[this.field];                               ❸
    let err = this.validator(value);
    if (err) data.errors.push(err);
    else data.message.push(this.field + ": " + value);
    return data;
  }
}
```

❶ Pass a parameter saying which part of the form we are going to validate.

❷ Collect the result (the email message to send) and the errors in the data and pass them to the next element.

❸ Select a field of the form to which to apply the logic of an element of the chain.

We can ultimately state the following:

- The form to validate (the problem) is partitioned; we specify which field we want to validate for each element of the chain.
- The result is built incrementally, so we get the form data, and we annotate it incrementally with the errors found or the final result (the email message we want to send).

We can then deploy our Chain of Responsibility reusing the actions we built previously with the simple command:

```
$ wsk action update pattern/validate --sequence \
  pattern/strategy-name,\
```

```
  pattern/strategy-email,\
  pattern/strategy-phone
  ok: updated action pattern/chainresp
```

Let's try a simple test on the command line:

```
$ wsk action invoke pattern/validate \
  -p name Michele -p email michele@sciabarra.com -r                    ❶
{
    "email": "michele@sciabarra.com",
    "errors": [                                                        ❷
        "missing phone"
    ],
    "message": [                                                       ❸
        "name: Michele",
        "email: michele@sciabarra.com"
    ],
    "name": "Michele"
}
```

❶ Missing the phone number.

❷ Error correctly detected.

❸ Parts of the form passing the validation check.

Here we show only a simple manual test case to be sure you have correctly deployed the code. We cover systematic testing in Chapter 6.

Command

The Command pattern provides a uniform interface to perform multiple tasks, with the task description encapsulated in a single object (see Figure 4-14). It works by:

1. Collecting all the information needed to perform a task in an instance of a Command interface
2. Sending that instance of the Command interface to an executor, for actually performing the task.

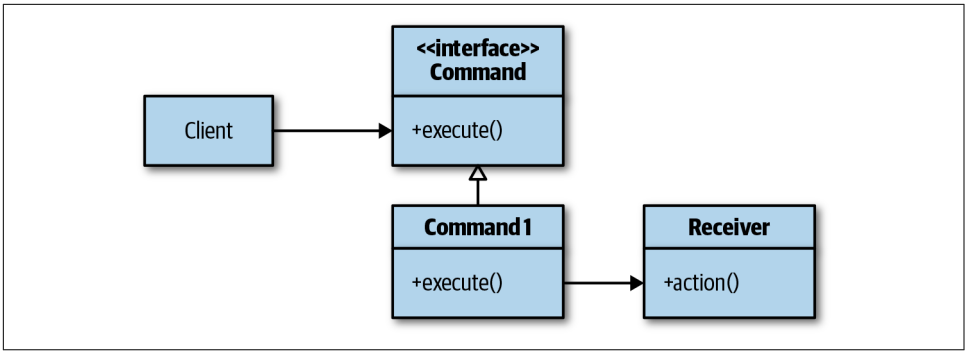


Figure 4-14. The Command pattern

In our contact form, we have a problem: managing a feed. A feed is, as we'll see later, an action that must execute some operation according to requests, called a `0`. As a result, we need to be able to store, retrieve, and delete data in a database (Figure 4-15).

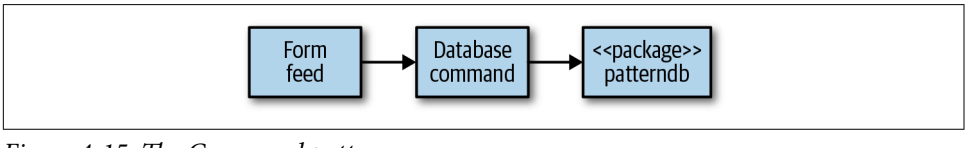


Figure 4-15. The Command pattern

We implement the data storage with the Command pattern. We define a command as a JSON object with the fields `command`, `key`, `value`, and `type`.

Strictly speaking, the pattern prescribes that you define a class whose instances are used to keep the command information. However, to implement it correctly we have to share the code for the `Command` class between the clients and the receivers. A common practice with OpenWhisk is to instead share data with JSON messages. We are hence not going to create a `Command` class. Instead, instances of actions just share the knowledge of the structure of the JSON. In this case, it wasn't necessary to enforce this, so our command is just a JSON structure with the three fields.

An object to represent the Command pattern in JavaScript is as follows:

```
{ "command": <CMD>,
  "type": <TYPE>
  "key": <KEY>,
  "value": <VALUE>,
}
```

We execute a different action for each value of *<CMD>*, most notably:

- `CREATE`, which saves the *<VALUE>* of the given *<TYPE>* in the database accessible with the *<KEY>*.
- `DELETE`, which deletes a record of the given *<TYPE>* accessible with the *<KEY>* (*<VALUE>* is ignored).
- `LIST`, which returns a list of each *<KEY>* and *<VALUE>* we stored in the database of the given *<TYPE>*.

Basically, the command implements a very simple key/value store: we save with a `key`, we retrieve the value with the `key`, and we can get a list of all the `key`s. However, we added a field `type` allowing us to store multiple key/value stores in the same database.

The structure of the action is as follows:

```
var kv = require("./lib/keyvalue.js")                    ❶

function main (args) {
  let command = args.command                             ❷
  let data = {
    type: args.type,
    key: args.key,
    value: args.value
  }
  switch (command) {
    case 'CREATE':                                       ❸
      return kv.create(data)
    case 'LIST':                                         ❹
      return kv.list(data)
    case 'DELETE':                                       ❺
      return kv.delete_(data)
  }
}
```

❶ Use the library *keyvalue.js*.

❷ Extract the `command` key.

❸ Create a record with the key and value.

❹ List all the records.

❺ Delete the record with the given value.

We used the name `delete_` with a final underscore to avoid conflicts with the `delete` JavaScript keyword.

Here we use a library called *keyvalue.js*, which is self-explanatory. In Chapter 5, we will go into more depth on database storage.

For now, let's execute an action at the command line:

```
$ wsk action invoke pattern/command-database \                    ❶
  -p command LIST -p type test -r
{"list":[]}
$ wsk action invoke pattern/command-database \                    ❷
  -p command CREATE -p type test \
  -p key alpha -p value 1 -r
{ "activationId": "994212b782224eb58212b782226eb561" }
$ wsk action invoke pattern/command-database \                    ❸
  -p command LIST -p type test -r
  {"list":[{"key":"alpha","value":1}]}
$ wsk action invoke pattern/command-database \                    ❹
  -p command DELETE -p type test -p key alpha
{ "activationId": "a93f1a0ada0f43edbf1a0ada0f53ed90" }
$ wsk action invoke pattern/command-database \                    ❺
  -p command LIST -p type test -r
{"list":[]}
```

❶ Check that the key/value store is empty.

❷ Create a key `alpha=1`.

❸ Now there is a value in the store.

❹ Delete the value using the key.

❺ Check that the key/value store is empty again.

Summary

In this chapter, we explored some simple design patterns in OpenWhisk. In particular, we explored a few patterns that are built into the design of OpenWhisk, and therefore used implicitly (Singleton, Facade, Prototype, and Decorator). We also explored some simple patterns that are frequently used with OpenWhisk (Strategy, Chain of Responsibility and Command).

CHAPTER 5

Integration Design Patterns in OpenWhisk

We have already explored some of the common and built-in patterns in OpenWhisk, which are useful for implementing single actions. In this chapter, we'll explore patterns that are useful for integrating different actions and interacting with users. We continue the implementation of some classical "Gang of Four" design patterns and also cover the ubiquitous Model-View-Controller pattern implemented for OpenWhisk.

In Figure 5-1 we can see the patterns covered in this chapter and the previous chapter:

1. Singleton
2. Facade
3. Prototype
4. Strategy
5. Chain of Responsibility
6. Command
7. Bridge
8. Proxy
9. Adapter
10. Observer
11. Composite and Visitor
12. MVC

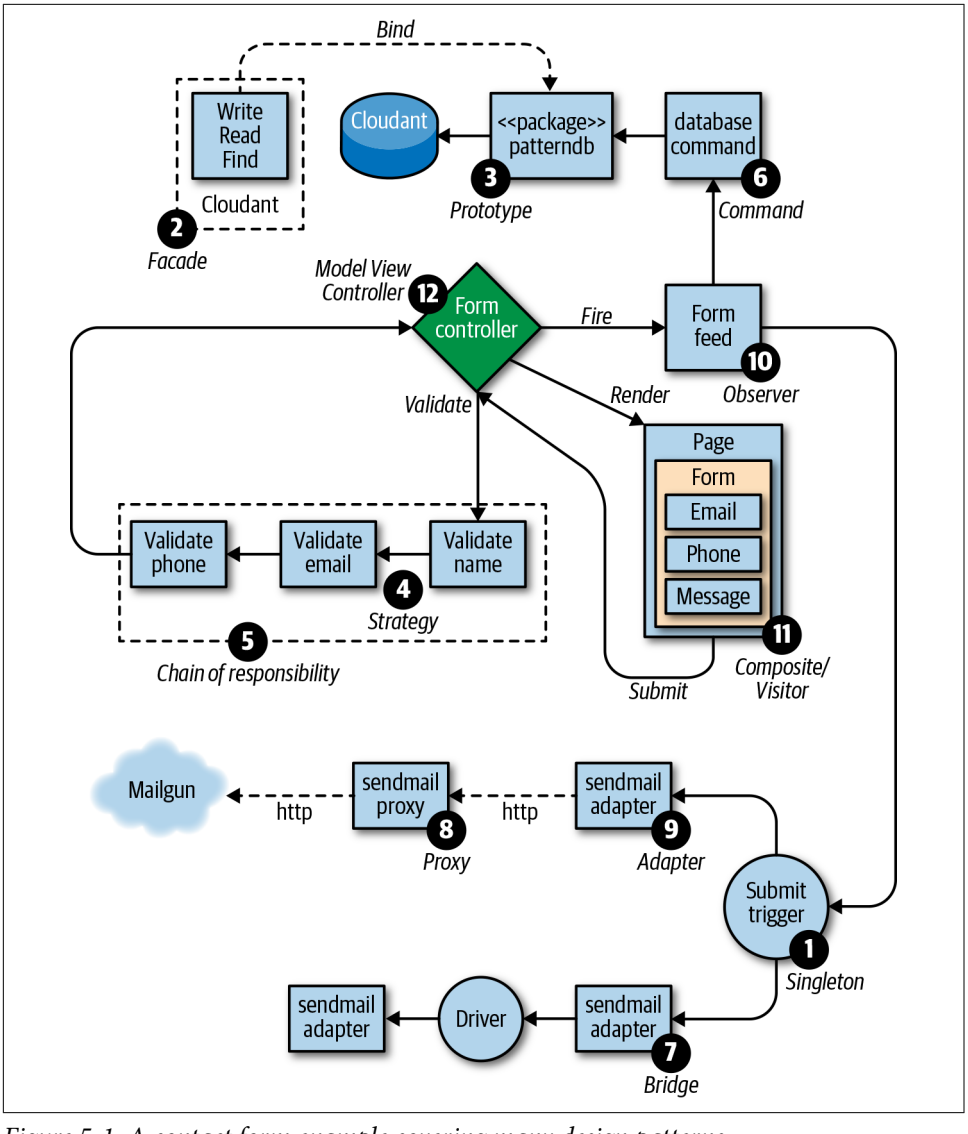


Figure 5-1. A contact form example covering many design patterns

The source code for the examples of the design patterns (this chapter and the previous chapter) are available in the GitHub repository.

Integration Patterns

In this section we discuss the patterns commonly used to connect different systems. These include:

Proxy
: Used when you want to keep the same interface to a system but change the details at the implementation level

Adapter
: Used for adapting requests from one interface to a system to another.

Bridge
: Used when you want to define a common generic interface to a class of systems and then specify different implementations for each case

Observer
: Used when you have a system producing events and many clients interested in those events who want to be notified of changes

Proxy

The Proxy pattern controls access and provides more functionality to an existing object. It works by creating an instance, a proxy, that is similar to the real object we want to access and is invoked in the same way. In the implementation of the proxy we perform the access control, or some functionality that otherwise changes the behavior; then we forward the request to the existing object (see Figure 5-2).

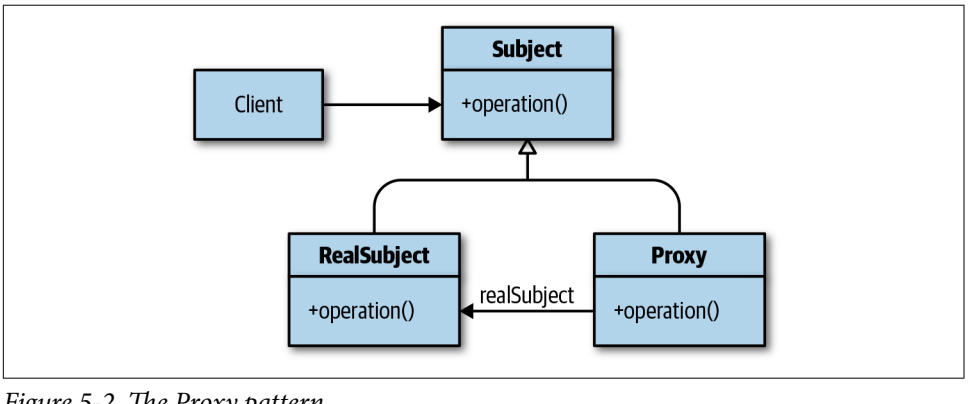


Figure 5-2. The Proxy pattern

For example, let's consider a simple proxy service that can send an email using an HTTP interface. In the example in Chapter 2, we used Mailgun to send emails. This service is accessible via HTTP, but it requires you to provide credentials before you

can send an email. Furthermore, the recipients are restricted to the email addresses specified when configuring the service.

To do better than this, we'll build an action, exposed as a public HTTP call, that can receive the subject and the body of an email, then handle the protocol and connect in a controlled and restricted way to Mailgun to deliver the message to the recipient we want (see Figure 5-3). We do this to provide a simplified and restricted access to a service, one of the common reasons that we use the proxy pattern.

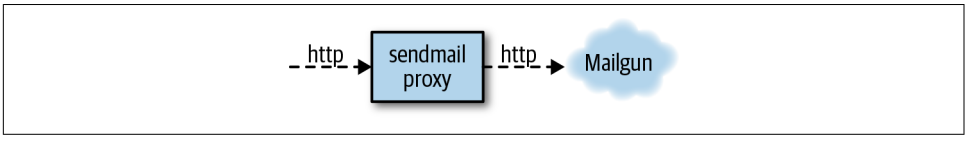


Figure 5-3. The Proxy pattern

Let's start from the configuration. First, we create a *sendmail.json* file like this:

```
{
  "mailuser": "<your-email@your-domain>",
  "maildomain": "<domain-from-mailgun>",
  "mailkey": "<mailgun-api-key>"
}
```

Here we use the configuration parameters of Mailgun described in Chapter 2. Next we deploy the `sendmail` action, described next, with:

```
wsk action create pattern/proxy-sendmail sendmail.js \
  -P sendmail.json --web true
```

The action receives the three parameters for accessing Mailgun, and two more parameters describing the email to send: `subject` and `body`.

Now we can write *sendmail.js*. We'll split the work of this task into two listings, as follows. The first one describes the configuration:

```
var http = require('https');                                    ❶
var querystring = require('querystring');

function main(args) {
  // Email
  var email = querystring.stringify({                           ❷
      'from' : args.mailuser,
      'to': args.mailuser,
      'subject': args.subject,
      'html': args.body
  });

  // Post Data
  var post_data = {                                             ❸
      host: 'api.mailgun.net',
      port: '443',
```

```
      path: '/v3/'+args.maildomain+"/messages",
      method: 'POST',
      headers: {
          'Content-Type':
   'application/x-www-form-urlencoded',
          'Authorization': 'Basic '+
    (new Buffer(args.mailkey).toString('base64')),
         'Content-Length': Buffer.byteLength(email)
      }
  };
  // <actual sending email>

}
```

❶ Use the native Node.js `https` module to send emails.

❷ Build the actual email (in HTML form format) using parameters provided by the action.

❸ Authenticate the access, providing a header for performing the HTTP Basic authentication.

We are simply accepting two fields, `subject` and `body`, adding the `from` and `to` fields (which happens to be same in our case), and including some headers to the HTTP call to perform authentication. Then we repeat the HTTP call as follows (this code replaces the `// <actual sending of email>` comment in the previous listing):

```
return new Promise(function(resolve, reject) {                    ❶
    // Set up the request
    var post_req = http.request(post_data,                        ❷
      function(res) {
        // send actual data
        res.setEncoding('utf8');
        res.on('data', function(data) {                           ❸
            resolve({ "result": data})
        });
    });
    // Handle errors
    post_req.on('error', function(err) {                          ❹
        console.log(err)
        reject({"error": err})
    })

    // Post the data
    post_req.write(email);                                        ❺
    post_req.end();
   })
```

❶ The `https` module is asynchronous, so we have to wrap it in a promise.

❷ Provide the `post_data` to the request.

❸ This is the returned value from Mailgun that we send as the answer to the proxy.

❹ Implement error handling.

❺ Insert the actual `email` body.

This listing executes the same `https` call that was used to invoke the proxy, changing some parameters. You invoke the proxy with the `https` protocol, and it emits another `https` call.

You can test the proxy by getting the URL and then invoking it with `curl` (note you are using an `http` interface here):

```
$ URL=$(wsk action get pattern/proxy-sendmail --url | tail -1)
$ echo $URL
https://openwhisk.eu-gb.bluemix.net/api/v1/web/ \
openwhisk@example.com_dev/pattern/proxy-sendmail
$ curl $URL'?subject=Hello&body=Hello+World'
```

If the parameters are correct, you will receive an email at the email address you configured.

Adapter

The Adapter pattern (Figure 5-4) accesses services using a different interface than the client requires. It works by providing an interface in the format the client expects, then implementing it in the format that the actual service uses.

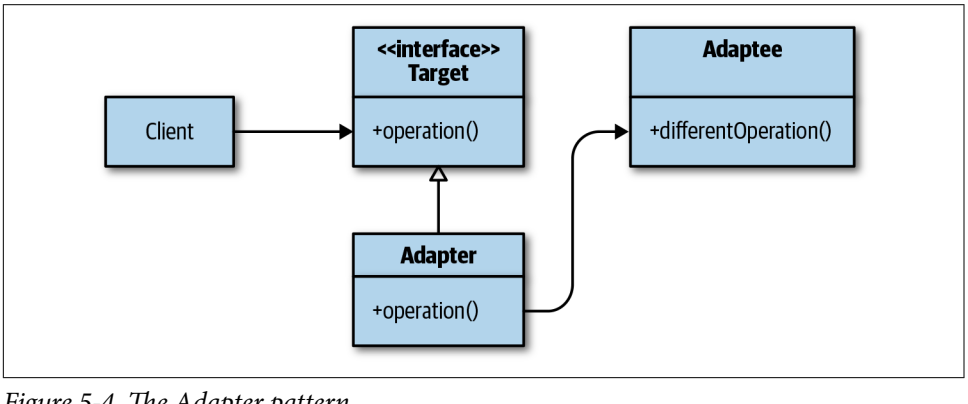


Figure 5-4. The Adapter pattern

Let's assume we have a service accessible only as an HTTP call. For example, in the previous example, we implemented a proxy action that can be invoked only in HTTP. Let's assume this proxy could not be invoked using the OpenWhisk API (actually, it can, but for the purposes of illustration we won't do this).

Let's assume we want to use the trigger we created for the Singleton pattern in the previous chapter to submit emails. A trigger primarily uses the OpenWhisk API to perform invocations of actions; it cannot use direct HTTPS calls.

So we have a service we can only invoke with HTTPS, but we want to use a trigger to submit data. To use a trigger we can only do an action invocation, which is ideal for the Adapter pattern, as shown in Figure 5-5.

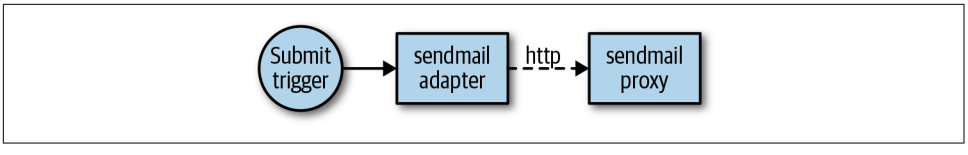


Figure 5-5. The Adapter pattern

We then write an action that receives data as a form invocation and forwards the request in HTTPS format to the proxy action.

The `form2http` action (in *form2http.js*) is as follows:

```
const https = require("https")

function main(args) {
    var message = `Name: ${args.name}<br>                                    ❶
Email: ${args.email}<br>
Phone: ${args.phone}`
    var body = encodeURIComponent(message)
    var query = "?subject=[Contact]&body="+body                              ❷

    return new Promise(function (resolve, reject) {                          ❸
        https.get(args.url+query, (resp) => {                                ❹
            resp.on('data', () => {})
            resp.on('end', () => resolve({result:"OK"}))                     ❺
        }).on("error", (err) => reject({error:err}))
    })
}
```

❶ Prepare the message to send from form data to URI format.

❷ Prepare a query string.

❸ Asynchronous call to be wrapped in a promise.

❹ Use the `args.url` parameter to locate the actual URL to invoke.

❺ `resolve` and `reject` invocations.

Since the URL to invoke is a parameter, we have to provide it when deploying the action.

We deploy as follows:

```
$ URL=$(wsk action get pattern/proxy-sendmail \
  --url | tail -1)                                    ❶
$ wsk action create pattern/adapter-form2http \
  -p url "$URL"                                       ❷
```

❶ Extract the URL of the target web action.

❷ Update the action with a parameter with the URL.

You can test to check if it sends an email with an action invocation:

```
$ wsk action invoke pattern/adapter-form2http -p messages \
  '["this","is", "a", "test"]' -r
{
    "result": "OK"
}
```

You should receive an email with the subject "[Contact]" and body "This is a test." Note this time we are using an invocation to send an email, so it's suitable to be hooked into a trigger.

Bridge

The Bridge pattern evolves abstractions and implements them independently. It works by defining a well-known abstraction that omits all the necessary implementation details for using a service. Implementations will provide the application-specific code, which can be evolved independently while keeping the abstraction valid (see Figure 5-6).

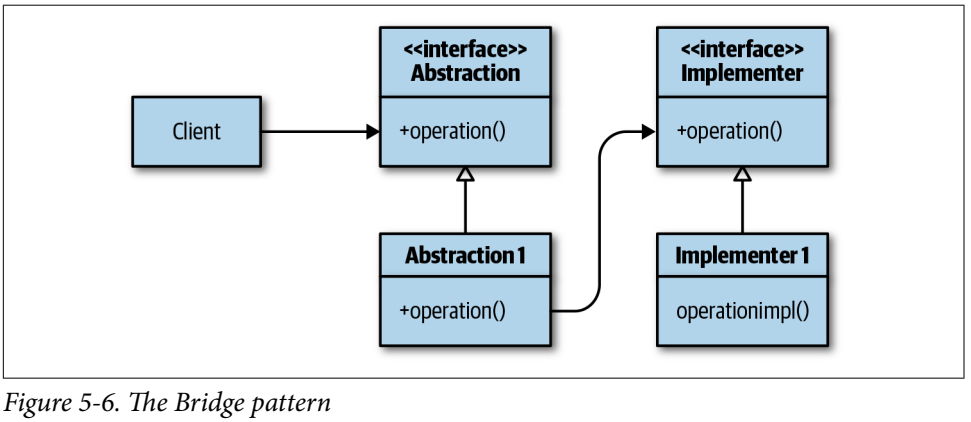


Figure 5-6. The Bridge pattern

For example, let's consider a common problem in OpenWhisk development: storing data. In our example, at some point we will have to preserve in some way the data of the contact form. We could simply implement an action that saves the data in the database. But all we need is a feature that, given some data, can store it in a database. We do not need to specify too many details of the actual database structure. Furthermore, we might even want to change the database. This is a good use case for the Bridge pattern (see Figure 5-7).

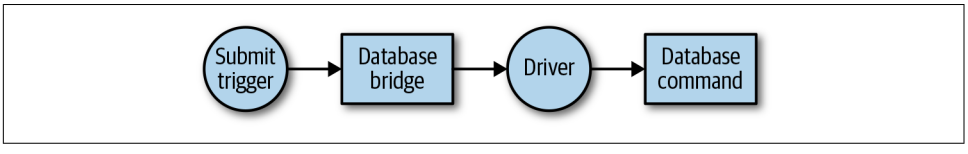


Figure 5-7. Demonstration of the Bridge pattern

We are going to decouple the operation of saving from the actual implementation. This way we can change the database used later.

In our example, the interface is the action itself; the implementation is in a module used as a library. Just changing the implementation of the module will provide access to a different database.

Here's the code for the `formsave` action:

```
const driver = require("./lib/driver")                    ❶

function main(args) {
    var key = args.email                                  ❷
    var value = {                                         ❸
        name: args.name,
        phone: args.phone
    }
    return driver.save(key, value)                        ❹
}
```

❶ Retrieve the driver implementation.

❷ Use the email as a key.

❸ Place the other two values in a map.

❹ Save it in a key/value store.

The implementation is actually leveraging the action we implemented for the Command pattern:

```
const openwhisk = require('openwhisk')

module.exports = {
```

```
  save: function (key, value) {
    return openwhisk().actions.invoke({
      name: 'pattern/command-database',                    ❶
      params: {
        command: 'CREATE',                                 ❷
        key: key,
        value: value,
        type: 'contact'                                    ❸
      }
    })
  }
}
```

❶ Invoke the action for the Command pattern.

❷ Add a new value with `CREATE`.

❸ Use `contact` as the type to store the data.

If we were going to use a different database, of course, the details would be different. But as long as we can keep an interface with a `save(key,value)` method we don't have to change the action, which means we can evolve the data storage solution independent of the interface of the action.

We can now test it with the following commands, first invoking the `formsave` action, then using the action in the Command pattern to see the result:

```
$ wsk action invoke pattern/bridge-formsave \
   -p email michele@sciabarra.com \
   -p name Michele -p phone 123456789 -r
{
    "activationId": "8fe95dfc17c04c63a95dfc17c0ec631d"
}
$ wsk action invoke pattern/command-database \
  -p command LIST -p type contact -r
{
    "list": [
        {
            "key": "michele@sciabarra.com",
            "value": {
                "name": "Michele",
                "phone": 123456789
            }
        }
    ]
}
```

Observer

The Observer pattern provides of events to a variable number of users.

It works as follows:

- The source of the event (a "subject") offers a registration (and deregistration) service, and it is able to keep track of the registered objects.
- When a user, hereafter called the "observer," needs notifications of an event, it registers itself with the subject. Note that the observer must use an interface known to the subject for notifications.
- When the event happens, the subject consults its recordings of the registered observers and invokes the known `update` operation for each observer, thus notifying them about the event.

From the point of view of the Observer pattern, a feed is an action implementing the observer. Your users are triggers requiring notifications about events sent to the action.

In this example, we implement the `pattern/observer-feed` action. To demonstrate how it works, I'll show in advance how to connect the feed to the trigger `pattern-singleton-submit` we use to process the form:

```
$ wsk trigger create pattern-singleton-submit --feed pattern/observer-feed
ok: invoked /_/pattern/observer-feed with id ca5956cbf4f040ca9956cbf4f0d0ca77
{
    "activationId": "ca5956cbf4f040ca9956cbf4f0d0ca77",
    "annotations": [...],                                                    ❶
    "duration": 92,
    "end": 1523023957025,
    "logs": [],
    "name": "observer-feed",
    "namespace": "openwhisk@example.com_dev",
    "publish": false,
    "response": { ... },                                                     ❷
    "start": 1523023956933,
    "subject": "openwhisk@example.com",
    "version": "0.0.13"
}
ok: created trigger pattern-singleton-submit
```

❶ Annotations removed.

❷ Response removed.

A feed, in practice, is just an action designed according to a protocol similar to the registration process used by the Observer pattern. A feed receives a request when you create, delete, pause, or unpause a trigger. Specifically, it receives an action invocation from the system when someone performs an action on the trigger and declares this action to be its feed. In particular, it receives the following parameters:

`lifecycleEvent`
: This can be `CREATE`, `DELETE`, `UPDATE`, `PAUSE`, or `UNPAUSE`.

`triggerName`
: This is the name of the trigger to be activated.

`authKey`
: This is the API key required to invoke the trigger from the outside.

Now let's implement the feed. We create the file *feed.js* with a structure very similar to the one we used for the Command pattern. We use `command` and `fire` here. We'll discuss them more later, but for now just note that `command` modifies the database using the Command pattern, while `fire` activates all the triggers whose names are saved in the database. A feed receives action invocations from the system when someone performs actions on the trigger, and declares this action to be its feed:

```
function main (args) {
  let event = args.lifecycleEvent
  if(event == 'CREATE') {                                    ❶
    return command('CREATE',
                   args.triggerName, args.authKey)
  } else if(event == 'DELETE') {                             ❷
    return command('DELETE',
                  args.triggerName)
  } else if(event == 'FIRE') {                               ❸
    return fire(args.value)
  } else {
    // not implemented PAUSE/UNPAUSE/UPDATE
    return {"error": "unimplemented "+event}                 ❹
  }
}
```

❶ Register a trigger.

❷ Unregister a trigger.

❸ Activate all the registered triggers.

❹ These events are not implemented, to keep the code simple.

We implemented the Observer pattern because the feed registers all the parties interested in an event. We save data to the database when we register and unregister triggers in the feed, as the pattern prescribes.

The Observer pattern prescribes a subject to notify the user of events, while the feed pattern and the `lifeCycle` request do not. To make the feed action a true observer, we added a `lifeCycle` event not required for the feed: `FIRE`.

Now let's see the functions we have used without defining. The code for `command` is pretty straightforward; it's just the `invoke` we already saw in our discussion of the API, and we use the key/value store implemented in the Command pattern:

```
function command(cmd, key = '', value = '') {
  let ow = openwhisk()
  return ow.actions.invoke({
    name: 'pattern/command-database',                    ❶
    result: true,
    blocking: true,
    params: {
      command: cmd,
      key: key,                                          ❷
      value: value,                                      ❸
      type: 'trigger'                                    ❹
    }
  })
}
```

❶ Use the `command-database` action to actually store data.

❷ The key is the trigger name.

❸ The value is the API key.

❹ Triggers are recorded as type `trigger`.

It is a little more complicated to notify interested parties about the events. We need to do the following:

- List all the registered triggers.
- Create API access with a specific API key for each trigger.
- Invoke the trigger, creating multiple promises.
- Collect the results and return them.

In code, it looks like this:

```
function fire(value) {
  return  command('LIST', 'trigger').then(res => {          ❶
    let promises = res.list.map(tr => {                     ❷
       let ow1 = openwhisk({ api_key: tr.value })           ❸
       return ow1.triggers.invoke({                         ❹
         name: tr.key,
         params: value
       })
     })
     return Promise.all(promises)                           ❺
     .then(results => ( {"results": results}))
     .catch(err => ({error: err}))
```

```
    })
}
```

❶ Use the `LIST` command from the database, listing all the registered keys of type `trigger`.

❷ Create a list of those invocations with `map` because each invocation is a promise.

❸ Create a different instance of the API because the API key can be different.

❹ Invoke the trigger.

❺ Wait for all the promises to complete and return the final result.

If we check the database, we'll see that there is now an additional record corresponding to the trigger.

Since we have a feed available, we can now register some rules for it. In particular, we are going to register rules to invoke the `bridge` action to save in the database and send an email when an event is triggered:

```
$ wsk rule update pattern-submit-adapter-form2http \
  pattern-singleton-submit \
  pattern/adapter-form2http
$ wsk rule update pattern-submit-brigde-formsave \
  pattern-singleton-submit \
  pattern/bridge-formsave
```

As a test, let's trigger the event with the following:

```
$ cat >data.json <<EOF                                          ❶
{
 "name":"Michele",
 "email":"michele@sciabarra.com",
 "phone":"1234567890"
}
EOF

$ wsk action invoke pattern/observer-feed \                     ❷
  -p lifecycleEvent FIRE \                                      ❸
  -p value "$(cat data.json)" -r                                ❹
{
  "results": [
    { "activationId": "397f9b16e95e42aebf9b16e95e52ae3" }
  ]
}
```

❶ Prepare a JSON file with the form data.

❷ Invoke all the registered triggers.

❸ `FIRE` is the event requesting the notification.

❹ Use the JSON file just created.

As a result of notifying users of the event, you should see an additional record in the database and receive an email.

User Interaction Patterns

In this section, we are going cover a pattern used to implement user interfaces in OpenWhisk known as the Model-View-Controller (MVC). The MVC pattern is widely used, and it has many variants; in fact, entire frameworks are designed around it.

In our case, we'll implement it by providing a "controller" action that will, in turn, invoke other actions both to handle the requested operations and to render the user interface (the "view" part of the MVC pattern). The "model" is generally the JSON data structure that is passed through the various actions.

To better understand views, first we'll discuss a couple of other patterns used to render them: the Composite and Visitor patterns.

These two patterns are covered in a slightly different way than the rest of the patterns in this chapter: the examples are not OpenWhisk-specific, mostly because implementing a composite as a combination of OpenWhisk actions is generally inefficient. But since these patterns are widely used for developing web applications, they are included here for completeness. An example is provided that can be used as part of an OpenWhisk application.

Composite

Some problems can be modeled by dividing them into smaller parts that are similar in structure to the central part. For example, an HTML page can be split into subparts that are still HTML pages. The Composite pattern models these types of problems by defining a common interface Component. Then you implement the other components as either *leaves*, which are components with no children, or *composites*, which can have other components as children.

The idea behind a composite is to create an object composed of parts similar to itself. It is a recursive structure like a tree. Each element of a composite has an operation and a variable number of children (Figure 5-8). It performs its operations with the help of the children, recursively. When a client invokes an operation on a composite, the composite invokes the operations on its children, who in turn invoke operations on their children, and so on until a "leaf" component with no children is reached and recursion ends.

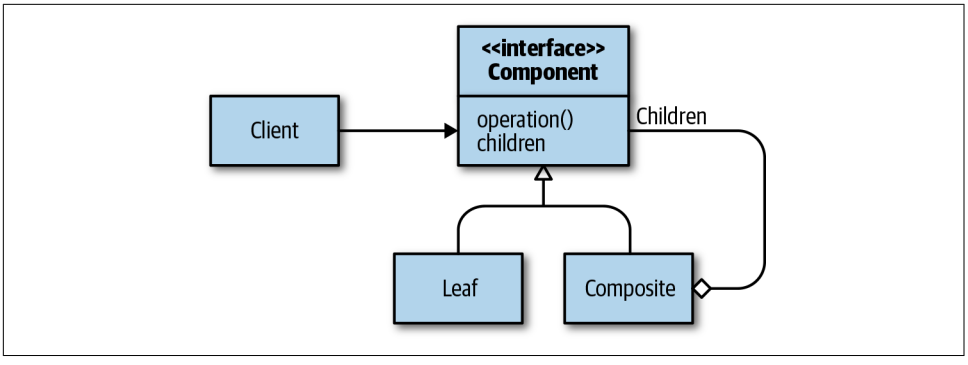


Figure 5-8. The Composite pattern

As an example of a composite, we'll provide an action that can render a web page split into components and subcomponents according to the Composite pattern.

We start by defining the Composite class as follows:

```
class Component {

    constructor(prefix, suffix = "") {
        this.prefix = prefix                    ❶
        this.suffix = suffix                    ❷
        this.children = []                      ❸
    }

    add(child) {
        this.children.push(child)               ❹
    }

  // <add here the code for rendering>
}
```

❶ A `Component` has a prefix.

❷ And an optional suffix.

❸ And an array of children, initially empty.

❹ Use this method to add children (another `Component`).

The idea is to model each snippet of HTML according to its nested structure. Some HTML tags do not nest, so you just create them with `let input = new Component('<input type="text">')`. Other tags do nest. For example, when you want to render a form, you need to start with `<form>` and end with `</form>`, and in the middle you place the HTML describing the fields of the form. So, you create a `Component`

with `let form = new Component('<form>', '</form>')`. To insert the `input` tag into the component, use `form.add(input)`.

Figure 5-9 shows the structure of the form, while the following code shows how it is rendered (note that the HTML is simplified):

```
const Component = require('./lib/component')

const name = new Component(`<input name="name">`)
const email = new Component(`<input name="email">`)
const phone = new Component(`<input name="phone">`)

const form = new Component(`<form>`, `</form>`)
form.add(name)
form.add(email)
form.add(phone)

const page = new Component(`<html>`, `</html>`)
page.add(form)
```

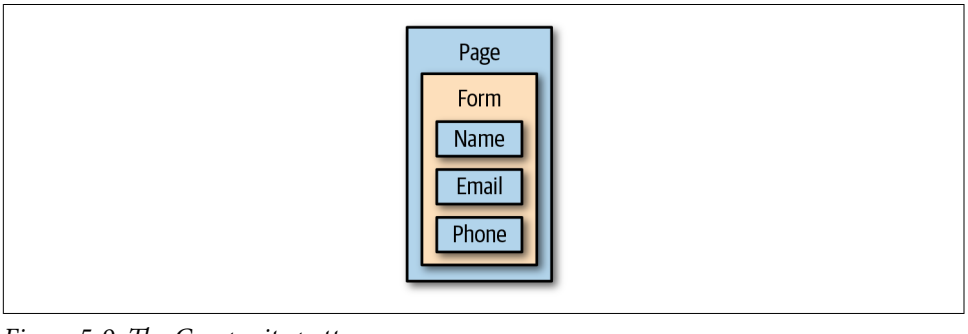


Figure 5-9. The Composite pattern

Now that we have our structure built, we need to render it. We will use the Visitor pattern for this, discussed next.

Visitor

The Visitor pattern performs arbitrary operations on complex and nested data structures with subcomponents (e.g., a composite). It works by implementing an `accept` operation for each element of the data structure that receives the visitor. The `accept` applies a `visit` of the Visitor pattern to itself and then applies the `accept` to each subcomponent (see Figure 5-10). This way, the visitor can visit all the components of the nested data structure.

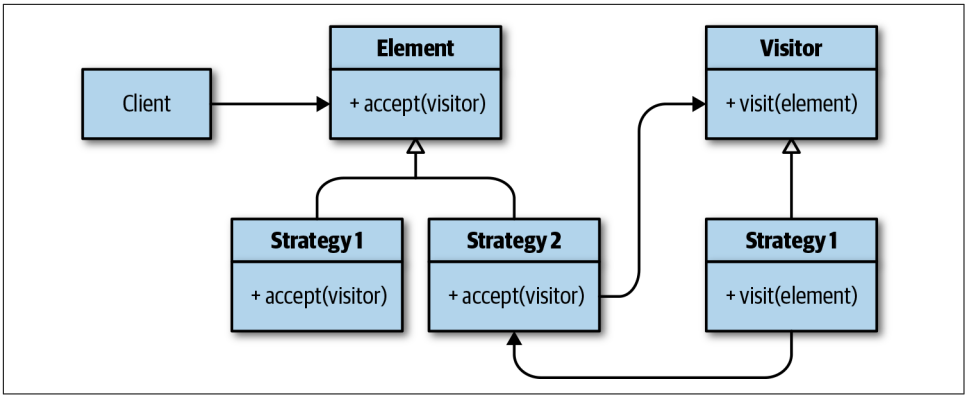


Figure 5-10. The Visitor pattern

The Visitor pattern is a good complement of the Composite pattern, since it is useful to write in HTML a representation of an HTML page we did in the preceding paragraph. We first implement a `Visitor` for the `Component` in the previous example class, as follows:

```
class Visitor {
    constructor() {                                          ❶
        this.prefixes = []
        this.suffixes = []
    }
    visit(item) {                                            ❷
        this.prefixes.push(item.prefix)
        this.suffixes.push(item.suffix)
    }
    render() {                                               ❸
        return this.prefixes.join("") +
               this.suffixes.reverse().join("")
    }
}
```

❶ The `Visitor` for the tree keeps a list of all the prefixes and suffixes.

❷ For each visited `Component`, the tree just records the prefix and the suffix in its list.

❸ Rendering is a matter of concatenating the prefix, and then the suffixes in reverse order.

To complete the example, we have to add the `accept` method to the `Component` class we defined earlier as follows:

```
accept(visitor) {
   visitor.visit(this)                                       ❶
```

```
    for(let child of this.children)                    ❷
        child.accept(visitor)                          ❸
}
```

❶ Visit the current composite.

❷ Iterate the children.

❸ Accept the visitor for each subcomponent.

Now we test what we get when applying our visitor to the composite from the preceding example:

```
let v = new Visitor()
page.accept(v)
console.log(v.prefixes)
console.log(v.suffixes)
console.log(v.render())
```

We get:

```
console.log view.test.js:
    [ '<html>',
      '<form>',
      '<input name="name">',
      '<input name="email">',
      '<input name="phone">' ]

  console.log view.test.js:
    [ '</html>', '</form>', '', '', '' ]
  console.log view.test.js:
    <html><form><input name="name"><input name="email"><input name="phone"></form></html>
```

This is the form defined with the composite as represented in HTML.

In Chapter 6 we cover how to write tests for actions without having to deploy the code in OpenWhisk. In addition to providing flexibility and creating better architectures, patterns also help segregate code, making it easier to test.

MVC

The Model-View-Controller (MVC) pattern is widely used for developing user interfaces. It reduces coupling between the rendering logic and control logic of the user interface by splitting the implementation of the user interface into three parts:

Model
: Primarily represents the current state of the user interface abstractly, omitting the more low-level details

View
: Reads the model and builds the actual user interface in a format you can render to the user

Controller
: Receives user interactions, determines the right course of action, and interfaces with the rest of the system

The three components interact in this way:

- The user interaction starts with the user looking at a rendering of the view and then interacting with it, producing events sent to the controller.
- The controller receives the events, consults the model, and then decides the state of the system after the user interaction, updating the model only.
- The model is then sent to the view again, which rerenders the user interface using the model to prepare the system to be ready for the next user interaction.

Then the loop restarts.

In the context of OpenWhisk, implementing the MVC pattern is pretty easy. In our contact form example, we implement a "controller" action and some "view" functions. The "model" is the JSON data structure that receives a request and returns an answer.

We already have some pieces we need to process the form:

- The form itself is rendered using the Composite/Visitor pattern (action: `pattern/composite-visitor-view`).
- The form validation is performed using the Chain of Responsibility pattern (action: `pattern/chainresp-validate`).
- When the form is validated, sending an email and saving to the database is performed by firing the feed (action: `pattern/observer-feed`), which in turn fires a trigger activating the actions we created to send an email and store data.

The controller ultimately needs to react to two possible actions of the user: the initial request (when the user opens the contact form) and the form submission. Because of how a web browser works, the initial request is translated to an HTTPS request using the method `GET`, while the form submission uses the HTTPS method `POST`. On the `POST` we trigger the validation, and then we decide what to do next, according to whether the form validates correctly or not. If it doesn't, we display the errors and repeat the form submission. If it does, we store the data and send an email.

Let's see the procedure in code:

```
function main (args) {
  let method = args.__ow_method                                    ❶
  let form = {                                                     ❷
    email: args.email,
    phone: args.phone,
    name: args.name
  }
  if (method == 'get') {                                           ❸
   return invoke('pattern/composite-visitor-view')
  } else {
    return invoke('pattern/chainresp-validate',                    ❹
                  form).then(result => {
      if (result.errors.length > 0) {                              ❺
        return viewErrors(result.errors)                           ❻
      } else {
        invoke('pattern/observer-feed', {                          ❼
          lifecycleEvent: "FIRE",
          value: form
        }).then(res => {
          console.log(res)
        })
        return viewOk(result.message)                              ❽
      }
    })
  }
}
```

❶ The method can be `GET` or `POST`.

❷ Collect the form data in a `form` object.

❸ Show the form.

❹ Validate the form.

❺ Check if there are errors.

❻ Display an error message.

❼ Form validated; send emails and save data.

❽ Display an OK message.

Let's now look at the helper methods, which are pretty simple and straightforward:

```
function invoke (name, args = {}) {                                ❶
  let ow = openwhisk()
  return ow.actions.invoke({
    name: name,
    blocking: true,
```

```
    result: true,
    params: args
  })
}

function viewErrors (errors) {                                    ❷
  return {
    body: '<h1>Errors!</h1><ul><li>' +
      errors.join('</li><li>') +
      '</li></ul><br><a href="javascript:window.history.back()">Back</a>'
  }
}

function viewOk (messages) {                                      ❸
  return {
    body: '<h1>Thank You!</h1><ul><li>' +
      messages.join('</li><li>') +
      '</li></ul>'
  }
}
```

❶ A wrapper to invoke an action in a simple way.

❷ Render an HTML answer with errors.

❸ Render an HTML answer with an "accepted" message.

Summary

In this chapter, we continued the exploration of design patterns we started in Chapter 4, considering more (and more complex) design patterns. In particular, we saw that the Proxy, Adapter, and Bridge patterns are used for connecting existing systems; the Observer pattern is key to implementing feeds; Composite and Visitor are useful for views; and MVC is widely used to structure interactions with the user.

CHAPTER 6

Unit Testing OpenWhisk Applications

For most newcomers, one of the biggest challenges when it comes to developing in a serverless environment is learning how to test and debug. Since you deploy your code continuously in the cloud, it's usually not possible to debug your code step by step. While a debugger can in some cases be useful, most developers split their applications into small pieces and test locally *before* sending the code to the cloud.

Once the application is tested in small pieces, it can be assembled and deployed as a whole. Then tests can be run against the final result, generally simulating user interaction, to ensure the pieces are working together correctly.

Testing in small parts is called *unit testing*, while simulating the interaction with the assembled application is called *integration testing*. In this chapter, we focus on how to do unit testing in OpenWhisk applications.

Luckily, there are plenty of unit testing tools available. You can use them to run small pieces of code on your local machine, then deploy only when the application is tested. Let's see how this works in practice.

In this chapter, we cover unit testing for the Node.js runtime. Unit testing for Python is covered in "Testing Python Actions" on page 173, and unit testing for Go is covered in "Testing Go Actions" on page 271.

The source code for this chapter's examples is available in the GitHub repository.

Using the Jest Test Runner

OpenWhisk applications generally run in the cloud. You write your code, deploy it, and then run it. However, to run unit tests, you need to be able to run your code on your local machine, without deploying in the cloud. In this chapter you are going to learn how to run action code locally to test them. But first you'll learn about Jest, a testing tool, and how to write tests with it.

Jest is one of many competing test tools in the JavaScript ecosystem. While it is the tool of choice for this chapter the techniques used here can be easily adapted to other testing tools.

Using Jest

Jest is a test tool developed by Facebook for testing React applications. We are not going to use React in this book, but Jest is independent of React, and it is a good fit for our purposes.

Jest is easy to install, fast, supports snapshot testing, and has extensive support for "mocking." Here's how to install it:

```
$ npm install -g jest                                    ❶
+ jest@22.4.3
updated 1 package in 13.631s
$ jest --version                                         ❷
v22.4.3
```

❶ Install Jest as a global command.

❷ Check if Jest is now available as a command.

Now that we have Jest installed, we can write a test. To begin, we will test a modified version of the word count action (*count.js*) we used in Chapter 3.

This action receives its input as a property `text` of an object `args`, splits the text into words, counts the words, and then returns a table containing all the words and how often they occur.

Here is the updated version, *wordcount.js*:

```
function main(args) {
    let words = args.text.split(" ")
    let map = {}
    let n = 0
    for(word of words) {
       n = map[word]
       map[word] = n ? n+1 : 1
```

```
    }
    return map
}
module.exports.main = main                                    ❶
```

❶ Add an export of the function.

In OpenWhisk, you can deploy a simple function. It works as long as it is a `main` function. When you want to use it locally in a test, however, it must be a proper module so a test can import it. That's why the last line is included in *wordcount.js*.

Always add the line `module.exports.main = main` at the end of all actions. It is required when you run a test or when you have to deploy the action in a zip file.

We can now write a test for `wordcount` that we can run with Jest. First, we'll write a `wordcount.test.js` file with the following code:

```
const wordcount =
      require("./wordcount").main                             ❶
test('wordcount simple', () => {                              ❷
    res = wordcount({text: "a b a"})                          ❸
    expect(res["a"]).toBe(2)                                  ❹
    expect(res["b"]).toBe(1)                                  ❺
})
```

❶ Import the module being tested.

❷ Declare a test.

❸ Invoke the function.

❹ Ensure it found two `a`s.

❺ Ensure it found one `b`.

We are ready to run the tests, but before we go on, we need to create a *package.json* file. This is a configuration file used by the Node Package Manager tool (`npm`) for storing package requirements and other information. We will use it for other functions later, but for now, we just need it to mark that this directory contains a Node.js project. It is as simple as this:

```
{ "name": "jest-samples" }
```

If you don't create a *package.json*, `jest` will walk up the directory hierarchy searching for one and will throw an error if it cannot find it. Once found, it will assume the directory it's in is the root directory containing a JavaScript application and it will look in all the subdirectories, searching for files ending in *.test.js*.

Now you can run the test. You can do this directly using the Jest command line from the directory containing all your files:

```
$ ls
package.json
wordcount.js
wordcount.test.js
$ jest
 PASS  ./wordcount.test.js
  ✓ wordcount simple (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.341s
Ran all test suites matching /wordcount/i.
```

As you can see, Jest started to search for tests, found our *wordcount.test.js* file, loaded it and executed the tests in it, then showed the results.

You usually do not run `jest` directly. A best practice is to configure a `test` command in the *package.json* file, as shown here:

```
{
  "name": "jest-samples",
  "scripts": {
    "test": "jest"
  }
}
```

Running Tests Locally

In general, actions in OpenWhisk run under Node.js, so the same code you execute locally with Node.js should also work with OpenWhisk as long as the environment is the same. To ensure the environment matches, when you run your tests you need to do the following:

- Run your tests with the same version of Node.js.
- Install the same Node.js packages locally.
- Load the code in the same way as OpenWhisk loads it.

- Provide the same environment variables that are available in OpenWhisk.

Let's discuss those needs in order.

Matching versions and packages

At the time of this writing, OpenWhisk provides different versions of the Node.js runtime; for example, there is one based on Node.js version 6.14.1 and another based on Node.js version 8.11.1. More recent versions might become available. Table 6-1 shows the preinstalled packages available for Node.js 6.

Table 6-1. Node.js 6 preinstalled packages

apn@2.1.2	async@2.1.4
body-parser@1.15.2	btoa@1.1.2
cheerio@0.22.0	cloudant@1.6.2
commander@2.9.0	consul@0.27.0
cookie-parser@1.4.3	cradle@0.7.1
errorhandler@1.5.0	express@4.14.0
express-session@1.14.2	glob@7.1.1
gm@1.23.0	lodash@4.17.2
log4js@0.6.38	iconv-lite@0.4.15
marked@0.3.6	merge@1.2.0
moment@2.17.0	mongodb@2.2.11
mustache@2.3.0	nano@6.2.0
node-uuid@1.4.7	nodemailer@2.6.4
oauth2-server@2.4.1	openwhisk@3.14.0
pkgcloud@1.4.0	process@0.11.9
pug@">=2.0.0-beta6 <2.0.1"	redis@2.6.3
request@2.79.0	request-promise@4.1.1
rimraf@2.5.4	semver@5.3.0
sendgrid@4.7.1	serve-favicon@2.3.2
socket.io@1.6.0	socket.io-client@1.6.0
superagent@3.0.0	swagger-tools@0.10.1
tmp@0.0.31	twilio@2.11.1
underscore@1.8.3	uuid@3.0.0
validator@6.1.0	watson-developer-cloud@2.29.0
when@3.7.7	winston@2.3.0
ws@1.1.1	xml2js@0.4.17
xmlhttprequest@1.8.0	yauzl@2.7.0

To prepare your environment for local testing, you need to install the same version of Node.js and the same packages available in OpenWhisk locally. You can do this using the Node Version Manager tool (`nvm`).

For example, let's assume you have an application running under the Node.js 6 runtime using the library `cheerio` for processing HTML. You can recreate the same environment locally using `nvm`.

`nvm` enables you to install a specific version of Node.js and easily switch among multiple versions on your machine. You can install it by following the instructions in the GitHub repository.

Once you have `nvm` installed, you can install the right environment for testing with the following commands (output shortened):

```
$ nvm install v6.14.1                                          ❶
Downloading https://nodejs.org/dist/\
  v6.14.1/node-v6.14.1-darwin-x64.tar.xz...
######### 100,0%
Now using node v6.14.1 (npm v3.10.10)
$ node -v
v6.14.1
$ npm init --yes                                               ❷
Wrote to  chapter4/package.json
$ npm install --save cheerio@0.22.0                            ❸
chapter4@1.0.0 chapter4
└─┬ cheerio@0.22.0
```

❶ Install the version of Node.js used in OpenWhisk.

❷ Create a *package.json* to store the package configuration.

❸ Install v0.22.0 of the `cheerio` library locally.

Loading the code

An action sent to OpenWhisk is a module, and you can load it with `require`. In general, OpenWhisk allows you to create an action without having to export if it is a single file, but in this case, you cannot test it locally because `require` wants you to export the module. Furthermore, if you're going to test functions internal to the module, you have to export them, too.

For example, let's reconsider the module we used to validate an email address in Chapter 4. Let's rewrite the file *email.js* in a form we can test locally. We will also make the error messages parametric (we will use this feature in another test later):

```
const Validator = require("./lib/validator.js")
function checkEmail(input) {                                    ❶
  var re = /\S+@\S+\.\S+/;
  return re.test(input)
}
var errmsg = " does not look like an email"
class EmailValidator extends Validator {
    validator(value) {
        let error = super.validator(value);
        if (error) return error;
        if(checkEmail(value))                                   ❷
            return "";
        return value+errmsg
    }
}
function main(args) {
    if(args.errmsg) {                                           ❸
        errmsg = args.errmsg
        delete args.errmsg
    }
    return new EmailValidator("email").validate(args)
}
module.exports = {
  main: main,                                                   ❹
  checkEmail: checkEmail                                        ❺
}
```

❶ Email validation logic is isolated in a function.

❷ Here we use the function to validate the email address.

❸ Parametric error message.

❹ The `main` function must always be exported.

❺ We will also test `checkEmail`.

Next we'll write *email.test.js*, step by step. First, we have to import the two functions we want to test from the module:

```
const main = require("./email").main
const checkEmail = require("./email").checkEmail
```

We can then write a test for the `checkEmail` function:

```
test("checkEmail", () => {
 expect(
  checkEmail("michele@sciabara.com"))
    .toBe(true)
 expect(
  checkEmail("http://michele.sciabarra.com"))
```

```
    .toBe(false)
})
```

Now let's add another test, testing instead the `main` function, the one we wanted to test in the first place:

```
test("validate email", () => {
 expect(main({email: "michele@sciabarra.com"})
   .message[0])
   .toBe('email: michele@sciabarra.com')
 expect(main({email:"michele.sciabarra.com"})
   .errors[0])
   .toBe('michele.sciabarra.com does not look like an email')
})
```

Setting environment variables

In OpenWhisk, you use the API to interact with other actions. We already discussed in Chapter 3 how you invoke actions, fire triggers, read activations, and so on. When you want to invoke other actions running in the same namespace as the main action, you need to use `require("openwhisk")` to access the API. At least, this is what happens when your application is running inside OpenWhisk.

Remember that when an action runs inside OpenWhisk, it has access to environment variables containing authentication information the API uses as credentials to execute the invocation. Be careful!

When your code is running outside OpenWhisk, as with unit testing, the OpenWhisk API cannot talk to other actions because every request needs authentication. Hence, if you try to run your code outside of OpenWhisk, you will get an error. We can demonstrate this with the simple action *dbread.js*, which can read a single record we put in the database with the contact form:

```
const openwhisk = require("openwhisk")
function main(args) {
  let ow = openwhisk()
  return ow.actions.invoke({
    name: "contactdb/read",
    result: true,
    blocking: true,
    params: {
      docid: "michele@sciabarra.com"
    }
  })
}
module.exports.main = main
```

If we deploy and run the action in OpenWhisk there are no problems:

```
$ wsk action update dbread dbread.js
ok: updated action dbread
$ wsk action invoke dbread -r
{
    "_id": "michele@sciabarra.com",
    "_rev": "5-011e075095301fcfc350cac66fd17c7e",
    "type": "contact",
    "value": {
        "name": "Michele",
        "phone": "1234567890"
    }
}
```

But let's try to write and run a test (*dbread.test.js*) for this simple action:

```
const main = require("./dbread").main
test("read record", () => {
   main().then(r => expect(r.value.name).toBe("Michele"))
})
```

If we run the test, the story is a bit different:

```
$ jest dbread
 FAIL  ./dbread.test.js
  ✕ read record (8ms)

  ● read record

    Invalid constructor options. Missing api_key parameter.

      1 | const openwhisk = require("openwhisk")
      2 | function main(args) {
    > 3 |   let ow = openwhisk()
      4 |   return ow.actions.invoke({
      5 |     name: "contactdb/read",
      6 |     result: true,

      at Client.parse_options (node_modules/openwhisk/lib/client.js:80:13)
      at new Client (node_modules/openwhisk/lib/client.js:60:25)
      at OpenWhisk (node_modules/openwhisk/lib/main.js:15:18)
      at main (dbread.js:3:12)
      at Object.<anonymous>.test (dbread.test.js:5:4)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        1.021s
Ran all test suites matching /dbread/i.
```

As we discussed in Chapter 3 when we introduced the API, the OpenWhisk library needs to know the host to contact to perform requests (the API host) and requires an authentication key to be able to interact with it.

When your application is running in the cloud in OpenWhisk, those keys are available through two environment variables, `__OW_API_HOST` and `__OW_API_KEY`. The OpenWhisk environment sets them before executing the code.

When your code is running locally, those variables are not available, unless your test code sets them. Luckily, it is it pretty easy to provide them locally. Indeed, if you are using the tool `wsk` and have configured it properly, it stores credentials in a file named *.wskprops* in the home directory. Because the format of this is compatible with the syntax of the shell, using bash you can load those environment variables with the following commands:

```
$ source ~/.wskprops
$ export __OW_API_HOST=$APIHOST
$ export __OW_API_KEY=$APIKEY
```

With the environment variables set, you can run this test successfully:

```
$ jest dbread
 PASS  ./dbread.test.js
  ✓ read record (22ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.951s, estimated 1s
Ran all test suites matching /dbread/i.
```

Your local code uses credentials and can contact and interact with the real server to run the test.

This test is *not* a unit test. It does not run entirely on your local machine. When you write unit tests, you should provide a local implementation that simulates the real servers. We will discuss in detail how to simulate OpenWhisk later in this chapter, when we talk about *mocking*.

For simplicity, if you want your environment to be correctly set up, I recommend leveraging the `npm test` command to run the tests and initialize them properly. If you add the following code into your `package.json` you can run tests with `npm test` instead of `jest`, without having to worry about setting up the environment variables manually:

```
"scripts": {
   "test": "source $HOME/.wskprops;                    ❶
__OW_API_HOST=$APIHOST
__OW_API_KEY=$AUTH jest"
 },
```

❶ This is actually one long line split into three lines for typesetting purposes.

Snapshot Testing

One of the downsides to testing is the burden of having to check results. You may end up writing a lot of code whose results you need to check against the expected values. Luckily, there are techniques (that we are going to see) to reduce the amount of repetitive code you have to write.

When you're testing, you choose a set of data corresponding to the various scenarios you want to verify. Then you write your test, invoking your code against the test data, and check the results. While defining the test data is generally fun, because you have to think about your code and what it does, verifying the results is not so fun.

Jest provides a large number of "matchers" to make checking results easier. Matchers work well when results are small, but sometimes test results are pretty large. In these cases you have to use *snapshot testing*.

Snapshot testing is pretty simple: when you run a test the first time, you save the test results in a file called a "snapshot." You can then make sure the result is correct. You do not have to write code—you only have to check the snapshot.

If the result is correct, you can commit the snapshot into the version control system. When you rerun a test, if there is already a snapshot in place, the test tool will automatically compare the new result of the test with the snapshot. If it is the same, the test passes; otherwise, it fails.

Let's see how this works in Jest using the simple matcher `toMatchSnapshot`. We will modify the two tests we just did to use snapshot testing. While the test for `checkEmail` simply checks if the result is `true` or `false` (so we don't have to modify it much to use snapshot testing), the test for `validateEmail` is a bit more complicated.

In that case, we wrote a test expression like this:

```
expect(main({email: "michele@sciabarra.com"})
   .message[0])
   .toBe('email: michele@sciabarra.com')
expect(main({email: "michele.sciabarra.com"})
   .errors[0])
   .toBe('michele.sciabarra.com does not look like an email')
```

Here, we are taking the test output and extracting some pieces (`.message[0]` or `.errors[0]`) to verify only a part of the result.

You could save yourself from the burden of thinking about which parts to inspect by just writing:

```
test("validate email with snapshot", () => {
  expect(main({email: "michele@sciabarra.com"})
   .toMatchSnapshot()
  expect(main({email: "michele.sciabarra.com"})
   .toMatchSnapshot()
})
```

Now, if you run the test, Jest will save the results as a snapshot. Of course, *always* make sure the snapshot is correct. Let's see the first execution of the test, when you create the snapshot:

```
$ jest email-snap
 PASS  ./email-snap.test.js
  ✓ validate email with snapshot (7ms)

 › 2 snapshots written.                                          ❶
Snapshot Summary
 › 2 snapshots written in 1 test suite.

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   2 added, 2 total                                    ❷
Time:        0.899s, estimated 1s
```

❶ Running the test creates two snapshots.

❷ Summary of the snapshots in the test.

Jest creates a folder named *snapshots* to store snapshots; then, for each test, it creates a file named after the filename containing the snapshots, like this:

snapshots/email-snap.test.js.snap

This file is designed to be human-readable, so you can inspect it to be sure the results in the snapshot are as expected.

For example, this is the content of the snapshot file just created:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`validate email with snapshot 1`] = `                    ❶
Object {
  "email": "michele@sciabarra.com",
  "errors": Array [],
  "message": Array [                                             ❷
    "email: michele@sciabarra.com",
  ],
}
`;

exports[`validate email with snapshot 2`] = `                    ❸
Object {
  "email": "michele.sciabarra.com",
  "errors": Array [                                              ❹
    "michele.sciabarra.com does not look like an email",
  ],
  "message": Array [],
}
`;
```

❶ Result of the first test, with a correct email address.

❷ We produce an array of messages.

❸ Result of the second test, with an incorrect email address.

❹ We produce an array of errors.

Once we know the snapshot is correct, all we need to do is save it in the version control system. If a snapshot is already present, running the test again will compare the test execution against the snapshot, as follows:

```
$ jest email-snap.test.js
 PASS  ./email-snap.test.js
  ✓ validate email with snapshot (7ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   2 passed, 2 total
Time:        0.876s, estimated 1s
```

Updating a snapshot

Once a snapshot is in place, Jest will make sure the test always returns the same result. But of course, things change. Let's look at an example of a test that fails because something in the snapshot changed, and then we'll see how to update the snapshot.

Let's assume that from now on we will only accept email addresses with a dot in the part before the `@`. Of course, we'd never do this, but right now we're trying to get our test to fail, so we'll use this as an example.

We change the regular expression to validate the emails as follows:

```
before: var re = /\S+@\S+\.\S+/;
after : var re = /\S+\.\S+@\S+\.\S+/;
```

Now, if we run the test again, it fails:

```
$ jest email-snap.test.js
FAIL ./email-snap.test.js
  ✕ validate email with snapshot (17ms)

  ● validate email with snapshot

    expect(value).toMatchSnapshot()

    Received value does not match stored snapshot 1.

    - Snapshot
    + Received

      Object {
        "email": "michele@sciabarra.com",
    -   "errors": Array [],
    -   "message": Array [
    -     "email: michele@sciabarra.com",
    +   "errors": Array [
    +     "michele@sciabarra.com does not look like an email",
        ],
    +   "message": Array [],
      }

      2 |
      3 | test("validate email with snapshot", () => {
    > 4 |     expect(main({email: "michele@sciabarra.com"})).toMatchSnapshot()
      5 |     expect(main({email: "michele.sciabarra.com"})).toMatchSnapshot()
      6 | })
      7 |

      at Object.<anonymous>.test (testing/strategy/email-snap.test.js:4:52)

 › 1 snapshot test failed.
Snapshot Summary
 › 1 snapshot test failed in 1 test suite. Inspect your code changes\
  or re-run jest with `-u` to update them.

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   1 failed, 1 passed, 2 total
```

```
Time:        0.967s, estimated 1s
Ran all test suites matching /email-snap.test.js/i.
```

We can fix the test replacing the first email:

```
expect(main({email: "michele.sciabarra@gmail.com"})).toMatchSnapshot()
```

However, if we simply run the test again, it will still be failing because the snapshot expects a `michele@sciabarra.com` somewhere.

Instead, we need to update the test using `jest -u`. The flag `-u` tells Jest to discard the current snapshot in the cache and re-create it.

After updating the test, of course, you should check that the new snapshot is correct. Indeed, we have now:

```
 "message": Array [
    "email: michele.sciabarra@gmail.com",
  ],
```

The snapshot can now be used for other tests.

Mocking

For a surprisingly large number of applications, you can run tests in isolation. However, at some point, while testing your code, you will have to interface with some outside service. For example, in OpenWhisk it is common to interact with other services via HTTP or by invoking the OpenWhisk API, perhaps to invoke other actions or activate triggers. This is where the boundaries of unit testing and integration testing start to blur.

Some developers would probably say it's impossible to test code that uses external APIs locally and that you can only test in the actual environment to get accurate results.

While this is partly true, it is possible to simulate "just enough" of the behavior of an external environment (or of any other system) by implementing a local "work-alike" of the remote system. We call this a *mock*. In our case, Jest explicitly supports mocking by replacing libraries with mocks we define.

We'll start with a simple example simulating an HTTP call. After that, we will cover in detail how to mock a significant part of the OpenWhisk API to simulate complex interactions between actions.

But first, let's learn a little bit more about what a mock really is.

What Is a Mock?

A mock is a piece of code that simulates the behavior of some non-local or complex service in a local environment for testing. For example, in the OpenWhisk environment, services that require mocking to be testable include:

- Action invocations
- HTTP requests
- Database access
- Messaging queues

This list is by no means exhaustive. Note that in a mock, you generally do not have to provide all the complexities of the remote service: you only deliver some results in response to specific requests.

In principle, you could mock everything, but in practice, this will probably require a lot of test code. So generally, you need to write your application code in a way that will isolate access to remote services in certain modules (that can be replaced by mocks) and use as little of the remote services as possible (in order to reduce the number of cases you have to test).

Mocking an HTTPS Request

To better understand how mocking works, let's consider a common problem: testing code that executes a remote HTTP call. To illustrate this, we are going to write a simple action, *httptime.js*, that returns the current time.

However, to make it "hard" to test, we will use another action, invoked via HTTP, that will return the date and time in full format, from which our action will extract the time.

To do this, we will have to use the Node.js `https` module and also wrap the call in a promise to conform to OpenWhisk requirements. For those reasons, the code is a bit more convoluted than we would like it to be. However, since mocking an HTTPS call is a frequent and important case, it is worth doing.

You can find the full example code in the GitHub repository.

An action to be tested by mocking

The code implements an action that returns the current time. It works by invoking via HTTPS another action that provides the date and time. It uses a promise-based API. In detail, it:

- Wraps everything in a promise, providing a `resolve` function
- Opens an HTTPS request to a URL, provided as a parameter of the action
- Defines event handlers for two events, `data` and `end`
- Collects data as it is received
- Extracts the time with a regular expression

Here is the `main` function:

```
const https = require("https")

function main(args) {
  return new Promise(resolve => {                          ❶
    let time = ""                                          ❷
    https.get(args.url, (resp) => {                        ❸
      resp.on('data', (data) => {                          ❹
        time += data                                       ❺
      })
      resp.on('end', () => {                               ❻
        var a = /(\d\d:\d\d:\d\d)/.exec(time)              ❼
        resolve({body:a[0]})                               ❽
      })
     })
   })
}
```

❶ The promise required to retrieve the results.

❷ This variable collects the input.

❸ Use the `https` module.

❹ Handle the `data` event (new data received).

❺ Collect the data, which may arrive in multiple chunks.

❻ Handle the `end` event (all data received).

❼ Extract the time from the date.

❽ Return the value extracted.

For convenience, let's first test in OpenWhisk. We need to create the action service that returns the current time and date. Note that in the following we use a shell feature to create an action without creating a file to store it:

```
$ CODE="function main() {\                                          ❶
>  return { body: new Date() } }"
$ wsk action update testing/now <(echo $CODE) \                     ❷
   --kind nodejs:6 --web true
$ URL=$(wsk action get testing/now --url | tail -1)                 ❸
$ curl $URL                                                         ❹
2018-05-02T19:42:34.289Z
```

❶ Store code in a variable.

❷ Use a bash feature to create a temporary file.

❸ Get the URL of the newly created action.

❹ Invoke the action via HTTP.

We can now deploy and invoke the action to see if it works:

```
$ wsk action update testing/httptime httptime.js \
  -p url "$URL" --web true
$ TIMEURL=$(wsk action get testing/httptime --url | tail -1)
$ curl $TIMEURL
20:06:55
```

Using a mock to test the action

The example action we just wrote is a typical example of an action that is difficult to unit test, for a few reasons:

- It invokes a remote service, so to run tests you need to be connected to the internet.
- You need to be sure the invoked service is readily available.
- The data returned changes every time, so you cannot compare the results with fixed values.

This is a perfect candidate for testing by mocking the remote service. We need to replace the `https` call with a mock that will not perform any remote call. Let see how to do that with Jest. Jest supports mocking through the ability to replace any JavaScript module with a mock without changing the code. In our case, to replace the `https` module with a mock we have to:

- Put the code that replaces the `https` module with our code in a directory called *mocks*.

- Use `jest.mock('https')` in our test code before importing the module to test.
- Write our test code as if we were using the real service.

The folder *mocks* must be in the same folder as *package.json*, a sibling to the *node_modules* folder.

This is the layout of the filesystem with the files we need for mocking:

```
httptime
├── package.json                    ❶
├── __mocks__                       ❷
│   └── https.js                    ❸
├── httptime.test.js                ❹
└── httptime.js                     ❺
```

❶ A placeholder file to declare it is a Node.js project (required by Jest).

❷ The directory containing the mocks.

❸ A mock to replace the `https` standard library.

❹ A few tests using the mock.

❺ The action to test by mocking.

When you perform a `require` in a Jest test, it will load your mocked code instead of the regular code. Since we want to replace the `https` module, we need to write the *mocks* code. For now, assume this code has already been written.

In the next test, pay attention to the fact that the data invoking the URL is the URL itself. In short, the mock of the URL `https://something` will return `something`! Since the only parameter we are passing to our mock is the URL, whose meaning is in our case irrelevant because we are not executing any remote calls, it makes sense to use the URL itself as the parameter to check the result.

Once the mock is in place, you can use it to write the test *https.test.js* as follows:

```
jest.mock('https')                                  ❶
const main = require('./httptime').main             ❷
test('https', () => {                               ❸
  main({
    url: '2000-01-01T00:00:00.000Z'                 ❹
  }).then(res => {                                  ❺
    expect(res.body).toBe('00:00:00')               ❻
```

```
  })
})
```

❶ Enable the mock.

❷ Import the action locally for testing.

❸ Declare a test.

❹ Invoke the `main` function—remember we use the URL as the data!

❺ The action returns a promise, so we have to handle it.

❻ Make sure the handler extracts the time from the date.

The `jest.mock` call is required only to replace built-in Node.js modules. In general, modules in the *mocks* directory will be used automatically before importing any module from *node_modules*.

To be sure, let's try to rerun the test with different data:

```
test('https2', () => {
  main({ url: '2018-05-02T19:42:34.289Z' })
  .then(res => {
    expect(res.body).toBe('19:42:34')
  })
})
```

Here's the final result:

```
$ jest httptime
 PASS  ./httptime.test.js
  ✓ https (5ms)
  ✓ https2 (1ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.222s
Ran all test suites matching /httptime/i.
```

Writing a mock for https

Now let's write the mock code that will replace the `https` call. To better understand the problem, let's consider what happens when the `https` module is invoked. As you can see, there are two steps:

```
https.get(URL, (resp) =>
   resp.on('data', (data) => {
     // COLLECT_DATA
   }
   resp.on('end', () => {
      // COMPLETE_REQUEST
   }
}
```

The "real" `https` module receives an object (`resp`) that acts as a gatherer of event-handling functions. There are two handlers required: one for each chunk of data that arrives, and another for when the request completes.

The HTTPS protocol, when performing a request, opens a connection to a URL, then reads the content and returns the results in multiple chunks. For this reason, you can have multiple calls to the `data` event executing the code marked as `COLLECT_DATA`. Hence, you need to collect the data before analyzing it. When all the data has been collected, you get one (and only one) invocation to execute the code marked as `COMPLETE_REQUEST`.

If we want to emulate this code with a mock, we need to perform the following steps:

1. Create an object that stores the event handler.
2. Invoke once (or more) the `data` event handler.
3. Invoke once the `end` event handler.

First, we declare an object that gathers the event handlers. This object must implement the method `on` as this is the method used to register event handlers:

```
var observer = {                                    ❶
   on(event, fun) {                                 ❷
       observer[event] = fun                        ❸
    }
}
```

❶ Create an object.

❷ Define an `on` function.

❸ Assign a function to a field; the field name is the event name.

This code looks complicated, but it's actually pretty simple. It defines an object that can register the event handlers and then can invoke them by name, as follows:

```
observer.on('something', doSomething)
observer.something(1)
```

So if you pass the observer to the `https.get()` function, it will register the event handlers. Once it returns, you will be able to invoke the two handlers registered by name.

Our action will register `resp.on('data', dataFunc)` and `resp.on('end', endFunc)`. Later, you will invoke them using `resp.data("some data")` and `resp.end()`.

Now that we have our observer, the full mock is much simpler to write. Remember what we want to happen:

1. Invoking the function `get`, we pass a handler object.
2. The handler register the functions to be invoked when data is received and the invocation complete.
3. The "magic" of our mock is that it returns the URL as the data.

We developed an `observer` that, once returned, can be used to invoke the user-defined functions `observer.data() and observer.end()`. So we pass it as handler, then we pass the `url` as data, and finally call `end`. In code it looks like this:

```
// observer here, omitted for brevity
function get(url, handler) {                    ❶
    handler(observer)
    observer.data(url)                          ❷
    observer.end()                              ❸
}
module.exports.get = get
```

❶ It will be invoked as `https.get(url, handler)`.

❷ Important: we are using the URL itself as the value returned by the mocked `https` call.

❸ Invoke the `end` handler.

Now the loop is closed and the tests written before can be implemented and run as shown.

Mocking the OpenWhisk API

Now that you understand testing with mocking, we can use it to test OpenWhisk actions using the OpenWhisk API without deploying them. In this section, we unit-test actions invoking other actions without deploying them.

The OpenWhisk API was covered in Chapter 3. Here, we develop a mocking library that can mock the more frequently used features of OpenWhisk: action invocation and sequences.

This library is not large, but its explanation can be complicated and you don't need to know all the details to use it in OpenWhisk, so here I'll only explain how to use it.

The library itself is available on GitHub along with the examples from this book. To use it in your tests, you need to install Jest and then download and place the file *openwhisk.js* in your *mocks* folder. You can then use this mocking library to write and run local unit tests that include action invocations and action sequences.

Using the Mocking Library to Invoke an Action

To use this library, we have to follow some conventions in the layout of our code. When your application runs in OpenWhisk, you do not have the problem of locating the code when invoking an action. You deploy an action with the name you choose, then use this name in the action invocation. It is the OpenWhisk runtime that resolves the names.

When you test your code locally and simulate invocations by mocking, you do not deploy your action code, but you still invoke it by name. So, our mocking library must be able to locate the actual code locally. It does this by following some conventions.

To illustrate these conventions, let's look at an example that invokes an action using the OpenWhisk API:

```
const ow = require('openwhisk')                                    ❶
test("invoke email validation", () => {
    ow()                                                           ❷
      .actions.invoke({                                            ❸
        name: "testing/strategy-email",                            ❹
        params: {
          email: "michele.sciabarra.com"                           ❺
        }
      }).then(res =>
        expect(res).toMatchSnapshot())
})
```

❶ Initialize the OpenWhisk library.

❷ Create the actual invocation instance.

❸ Perform the invocation.

❹ Action name.

❺ Parameters.

You must translate the name `testing/strategy-email` into the name of a file in the local filesystem. We use as the base in the filesystem the folder containing *package.json* and *node_modules*, which is the project root.

By convention, we name our actions according to this structure:

```
<package>/<prefix>-<action>
```

Then we expect the action code to be placed in a file named:

- *<package>/<prefix>/<file>.js*

When we test the code locally by invoking the `testing/strategy-email` action, the mocking library works like this:

- The `require("openwhisk")` will load the *mocks*/openwhisk.js library.
- It can locate its position from the `__dirname` variable, and hence find the project root (i.e., the parent directory).
- Using the name of the action, it can now locate the code of the action to invoke: in our case, *<project-root>/testing/strategy/email.js*.

Mocking Action Parameters

There is another essential feature we need to complete our simulation for testing. In OpenWhisk, actions can have additional parameters. Those parameters can be specified when you deploy the action or can be inherited from the package to which the action belongs. When we use our mocking library, we can simulate those parameters by adding a file with the same name as the action and the extension *.json*.

For example, in our case, we have the file *testing/strategy/email.json* in the same folder as *email.js* with the content:

```
{
    "errmsg": " is not an email address"
}
```

The mocking library uses this file, preloading the parameters and passing them as args. Indeed, we can check the test has used the parameter `errmsg`. We need to check the snapshot to see:

```
$ cat __snapshots__/invoke.test.js.snap                                    ❶
// Jest Snapshot v1, https://goo.gl/fbAQLP
exports[`invoke email validation 1`] = `
Object {
  "email": "michele.sciabarra.com",
  "errors": Array [
    "michele.sciabarra.com is not an email address",                       ❷
  ],
  "message": Array [],
}
`;
```

❶ Dump the snapshot file the terminal.

❷ The error message produced is the one specified in the *email.json* file.

Mocking a Sequence

We can now complete our analysis of the mocking library by describing how to mock a sequence. In Chapter 5 we saw how the Chain of Responsibility pattern is implemented as a sequence. A sequence does not have a corresponding action, but it does exist as a deployment declaration. We can simulate an action sequence with a particular value in the JSON file we use to pass the parameters.

For example, let's consider the following test sequence:

```
test('validate4', () =>
 ow().actions.invoke({
   name: 'testing/chainresp-validate',
   params: {
     name: 'Michele',
     email: 'michele.sciabarra.com',
     phone: '1234567890'
   }
}).then(res => expect(res).toMatchSnapshot()))
```

There is not a *testing/chainresp/validate.js* file, but there is a *testing/chainresp/validate.json* with this content:

```
{
  "__sequence__": [
    "testing/strategy-name",
    "testing/strategy-email",
    "testing/strategy-phone"
  ]
}
```

We can now write a test as follows:

```
const ow = require('openwhisk')
test('validate', () =>
  ow().actions
    .invoke({
      name: 'testing/chainresp-validate',
      params: {
        name: 'Michele',
        email: 'michele.sciabarra.com',
        phone: '1234567890'
      }
    })
    .then(res => expect(res).toMatchSnapshot()))
```

The mocking library then reads the `__sequence__` property from *validate.json* and translates in a sequence of invocations (as it would happen in the real OpenWhisk), allowing us to test the result of a chained invocation locally.

We can see that the mocking of the sequence works by inspecting the snapshot:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP
exports[`validate 1`] = `
Object {
  "email": "michele.sciabarra.com",
  "errors": Array [
    "michele.sciabarra.com is not an email address",                    (1)
  ],
  "message": Array [
    "name: Michele",
    "phone: 1234567890",
  ],
  "name": "Michele",
  "phone": "1234567890",
}
`;
```

(1) Error message defined as a parameter.

Summary

In this chapter you learned about testing and debugging JavaScript applications running in OpenWhisk. First, we gathered information and prepared a local environment that mimics OpenWhisk, and we installed a test runner (Jest). Then we went through code for testing actions, either locally or by connecting to OpenWhisk, with particular emphasis on verifying the results through snapshot testing. Finally, we covered an important technique used to test code that connects to other code: mocking.

PART II

Advanced OpenWhisk Development

In this second part, you'll take your OpenWhisk skills a little further. We'll use two more programming languages to develop OpenWhisk actions: Python and Go. We'll analyze how to create, deploy, and test OpenWhisk actions in Python, and create a simple CRUD application. Then we'll do the same for Go actions, and to create a simple web chat application. We'll also explore two important components of OpenWhisk applications: the NoSQL database CouchDB and the messaging queue Kafka. Finally, we complete the book by exploring how to install OpenWhisk in the cloud and on-premises using Kubernetes.

CHAPTER 7

Developing OpenWhisk Actions in Python

In this chapter, you will learn how to write OpenWhisk actions using Python.

While knowledge of Python is a prerequisite to fully understand the examples in this chapter, the code should be comprehensible to most developers as Python is one of the easiest and most readable programming languages around. Still, I recommend that you check out the Python tutorial before reading this chapter and Chapter 9 if you haven't worked with Python before.

The source code for this chapter's examples is available in the GitHub repository.

The Python Runtime

Let's start by exploring the Python runtime. As you'll see, it executes actions similar to the examples we saw earlier.

You develop your actions by creating a function, `main`, that will receive a dictionary as input and must also return a dictionary as output (you can specify a different function name if you want).

For example:

```
def main(args):                                          ❶
    name = args.get("name", "world")                     ❷
    greeting = "Hello " + name + "!"
    print(greeting)                                      ❸
    return {"hello": greeting }                          ❹
```

❶ The entry point is a function with a dictionary as a parameter.

❷ You access the fields of this dictionary using Python functions like `get`.

❸ What you print goes in the log.

❹ The returned value must be a dictionary too.

Deploying the action and invoking it works the same way as it does in JavaScript:

```
$ wsk action update python/hello hello.py
ok: updated action python/hello
$ wsk action invoke python/hello -p name Mike -r
{
    "hello": "Hello Mike!"
}
```

Here, we did not specify which programming language to use, but the CLI is smart enough to infer that the action requires the Python runtime. The default is Python 2, but if you want to use Python 3 use the option `--kind python:3`.

What's in the Python Runtime?

Now that we have our shiny new runtime, let's use some inspection actions to take a closer look.

The first step is to write a script that prints the version:

```
import sys
def main(args):
    return { "version": sys.version}
```

Let's deploy and run it:

```
$ wsk action update python/version version.py
ok: updated action python/version
$ wsk action invoke python/version -r
{
    "version": "2.7.15 (default, Sep 12 2018, 02:38:23) \n[GCC 6.4.0]"
}
```

We can see that by default it uses Python 2, but let's specify that we want Python 3, instead:

```
$ wsk action update python/version3 \
  version.py --kind python:3
ok: updated action python/version3
wsk action invoke python/version3 -r
{
```

```
        "version": "3.6.6 (default, Sep 12 2018, 02:15:29) \n[GCC 6.4.0]"
    }
```

Now let's create a script that can produce a listing of all the available libraries in the runtime in the format *<library>==<version>*. The standard *requirements.txt* file, used by the `pip` package manager, actually uses this format; we use it later to create a test environment.

In Python, the `pkg_resources` library provides a list of the third-party packages installed and writes a web action that will output our file:

```
import pkg_resources                                          ❶
def main(args):
    requirements = ""
    for d in pkg_resources.working_set:                       ❷
        requirements += d.project_name                        ❸
        requirements += "=="
        requirements += d.version
        requirements += "\n"
    return {
        "body": requirements                                  ❹
    }
```

❶ This package gives access to a list of available packages.

❷ This object contains all the available package information.

❸ We extract just the package name and version.

❹ We return a web action output.

Let's now use this script to extract *requirements.txt*, which we can use later to rebuild the runtime environment locally:

```
$ wsk action update python/requirements requirements.py --web true
ok: updated action python/requirements
$ curl $(wsk action get python/requirements --url | tail -1)
wsgiref==0.1.2
Python==2.7.15
zope.interface==4.5.0
wheel==0.31.1
...
```

Libraries Available in the Runtime

Table 7-1 lists the libraries included in the runtime for Python 2 at the time of writing.

Table 7-1. Python 2 action runtime package list

Package	Version
Flask	0.11.1
Jinja2	2.10
MarkupSafe	1.0
PyDispatcher	2.0.5
Python	2.7.15
Scrapy	1.1.2
Twisted	16.4.0
Werkzeug	0.14.1
asn1crypto	0.24.0
attrs	18.2.0
beautifulsoup4	4.5.1
cffi	1.11.5
click	6.7
cryptography	2.3.1
cssselect	1.0.3
enum34	1.1.6
functools32	3.2.3.post2
gevent	1.1.2
greenlet	0.4.15
httplib2	0.9.2
idna	2.7
ipaddress	1.0.22
itsdangerous	0.24
kafka-python	1.3.1
lxml	3.6.4
parsel	1.5.0
pip	18.0
pyOpenSSL	18.0.0
pyasn1	0.4.4
pyasn1-modules	0.2.2
pycparser	2.18
python-dateutil	2.5.3
queuelib	1.5.0
requests	2.11.1
service-identity	17.0.0
setuptools	40.4.1

Package	Version
simplejson	3.8.2
six	1.11.0
virtualenv	15.1.0
w3lib	1.19.0
wheel	0.31.1
wsgiref	0.1.2
zope.interface	4.5.0

Table 7-2 lists the available packages in the Python 3 runtime at the time of writing.

Table 7-2. Python 3 action runtime package list

Package	Version
Automat	0.7.0
Flask	0.12
Jinja2	2.10
MarkupSafe	1.0
PyDispatcher	2.0.5
Scrapy	1.3.3
Twisted	17.1.0
Werkzeug	0.14.1
asn1crypto	0.24.0
attrs	18.2.0
beautifulsoup4	4.5.3
cffi	1.11.5
click	6.7
constantly	15.1.0
cryptography	2.3.1
cssselect	1.0.3
gevent	1.2.1
greenlet	0.4.15
httplib2	0.10.3
idna	2.7
incremental	17.5.0
itsdangerous	0.24
kafka-python	1.3.4
lxml	3.7.3
parsel	1.5.0

Package	Version
pip	18.0
pyOpenSSL	18.0.0
pyasn1	0.4.4
pyasn1-modules	0.2.2
pycparser	2.18
python-dateutil	2.6.0
queuelib	1.5.0
requests	2.13.0
service-identity	17.0.0
setuptools	40.3.0
simplejson	3.10.0
six	1.11.0
virtualenv	15.1.0
w3lib	1.19.0
wheel	0.31.1
zope.interface	4.5.0

Since the actual package versions may vary, it is best that you use the scripts here to see what you have available. I don't expect the packages to change drastically in future releases, but developers periodically refresh the runtimes, updating the versions of the packages.

Using Third-Party Libraries

The examples so far have only used a single Python file. But most Python programs are composed of many files and can include third-party libraries. Next, we will go over how to deal with those requirements and deploy multifile actions in OpenWhisk.

Packaging a Python Application in a Zip File

To deploy an action composed of multiple files, you need to build a zip file that conforms to some rules.

The most important rule is that it must contain, at the top level (*not* in a subdirectory), a file called *__main__.py*. In this file, you have to define a function whose name is the one specified with the flag `--main`. If you omit the value, the default name is `main`.

To illustrate, let's create a *__main__.py* file containing a function called `main`:

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

We can now build a zip file and deploy it and test it:

```
$ zip zipfile.zip __main__.py                                    ❶
  adding: __main__.py (deflated 47%)
$ wsk package update python                                      ❷
ok: updated package python
$ wsk action update python/zipfile \                            ❸
      zipfile.zip --kind python:3
ok: updated action python/zipfile
$ wsk action invoke python/zipfile -r                            ❹
{
    "main": "Welcome stranger"
}
```

❶ Package the action in a zip file.

❷ Make sure we have a `python` package.

❸ Deploy the action as a zip file.

❹ Invoke it and see the result.

The action does not have to be named `main`. If we have, for example, this *__main__.py* including this `hello` function

```
def hello(args):
    return {
        "hello": "Hello %s" %
           args.get("name", "world")
    }
```

we can build the zip in the same way as before, but we have to deploy it in a slightly different way:

```
$ wsk action update \
 python/zipfile-hello zipfile.zip \
  --kind python:3 --main hello                                   ❶
ok: updated action python/zipfile-hello
$ wsk action invoke python/zipfile-hello -r
{
    "hello": "Hello world"
}
```

❶ Note here the use of `--main`.

Of course, a zip file comprising only one file is pointless. Let's create a Python module, stored in the file *hi.py*, that can we deploy in the same zip file:

```
# File: hi.py
def hi(name):                                                    ❶
    if name:
        return "Hi %s" % name
    return "Hi all!"

# File: __main__.py
import hi                                                        ❷
def main(args):
    return {"hi":
        hi.hi(args.get("name"))}                                 ❸
```

❶ The function `hi` is stored in a different Python module.

❷ We import the module `hi` in the `main` function.

❸ Refers to the function in the module.

In Python it is common to collect functions in a module and import them in another. But now we have to deploy both files:

```
$ zip zipfile-hi.zip __main__.py hi.py
  adding: __main__.py (deflated 47%)
  adding: hi.py (deflated 30%)
$ wsk action update python/zipfile-hi \
   zipfile.zip --kind python:3
ok: updated action python/zipfile-hi
$ wsk action invoke python/zipfile-hi -r
{
    "hi": "Hi all!"
}
```

Using virtualenv

As we have seen, there are many libraries available out of the box in the OpenWhisk Python runtimes. But what do we do when we want to use libraries that are not in the default set? Luckily, we can include additional third-party libraries in the zip file we use to deploy multifile actions.

Python has a vast repository of open source libraries available in a common registry called the Python Package Index (PyPI), which is accessible with the tool `pip`. You can browse the available packages at *https://pypi.org*.

The inclusion of additional libraries requires the use of two Python-specific tools: `pip` and `virtualenv`.

`pip` (short for "Pip install packages"--a recursive acronym, a common pun in programming) is the standard Python package manager, included in recent distributions of the language. In a normal Python environment, when you need a package that is available in PyPi, you install it with the command `pip <package>`.

Unfortunately, it is not as simple to install the package in a serverless environment. This is because we have a specific version of Python within our runtime, and a particular set of libraries already installed, with exact versions.

When we install a package with `pip`, it can have dependencies. As a result, `pip` also installs all the dependencies, and the dependencies of the dependencies, but skips those dependencies already available in our environment. Things can become even more complicated when we consider versions: `pip` can pick a specific version of the library we have requested, then update some other libraries available in the environment if the library asks for a more recent version of an already-present package…

In practice, we need to be able to run our `pip` installer in the same environment we will have in the runtime image, then upgrade it, and finally, send the final environment. How can this be done?

Here is where the utility `virtualenv` comes in. `virtualenv` can create a fresh copy of a given environment that can then be modified or updated, without affecting the original environment. The "virtual environment" is a subdirectory with many folders and links to the existing environment. The `virtualenv` command creates it. Once we have this virtual environment, we can activate it and install additional packages (with `pip`) in it.

But here's the problem: we have to do that using the same environment that is in the runtime. Luckily, those environments are Docker images we can pull and use to build the virtual environment that we then send to OpenWhisk. Let's look at some examples.

How Virtualenv and Pip Work

First, let's see how Virtualenv and Pip work by installing the `yattag` package. The following steps can be done in a single command.

This is the listing (with some of the more verbose and irrelevant output eliminated):

```
$ docker run --rm -ti  openwhisk/python3action bash                    ❶
latest: Pulling from openwhisk/python3action
# python -c 'import yattag'                                            ❷
ModuleNotFoundError: No module named 'yattag'
# cd /tmp
# virtualenv virtualenv                                                ❸
Using base prefix '/usr/local'
New python executable in /tmp/virtualenv/bin/python
Installing setuptools, pip, wheel...done.
```

```
# source virtualenv/bin/activate                                    ❹
(virtualenv) # pip install yattag                                   ❺
Collecting yattag
Successfully built yattag
Installing collected packages: yattag
Successfully installed yattag-1.10.0
# python -c "import yattag"                                         ❻
```

❶ Enter the Python runtime.

❷ Check if there is a `yattag` module; if not, error.

❸ Create a virtualenv in */tmp/virtualenv*

❹ Activate it.

❺ Install the `yattag` module with `pip`.

❻ Success! No errors.

Automating the Virtual Environment

Now that the procedure is clear we can automate it, with the help of Docker. We will perform the following steps:

Deploy and run.

We need to use the Docker container to build the virtualenv, but we have to store the result in a directory outside the Docker container. The `-v` (volume) switch in Docker allows mapping our current folder into a folder inside the Docker container.

Using the yattag Library

Let's assume we want to write an action in Python to produce an HTML result—i.e., a web action. The usual way is to embed the HTML in the code, but this generally leads to code that is difficult to read and maintain. A better option would be to describe the HTML in Python.

`yattag` is a library designed for this purpose. For example, we can write a simple `hello world` Python action that produces HTML as follows:

```
from yattag import Doc
doc, tag, text = Doc().tagtext()                                    ❶

def main(args):
  with tag("html"):                                                 ❷
    with tag("body"):
```

```
    with tag("h1"):
      text("Hello %s" %                                    ❸
         args.get("name", "world"))
  return {
    "body": doc.getvalue()                                 ❹
  }
```

❶ Import the library to generate the HTML.

❷ `with tag` represents an HTML tag.

❸ We use `text` to produce text output in the HTML.

❹ Finally, we return the HTML we built as a Python dictionary.

Building the Virtualenv, Including a Library

Before building the virtual environment, we create the *requirements.txt* file that can be used to give `pip` a list of the required libraries. With it, we can use `pip install -r requirements.txt` instead of specifying all the libraries on the command line.

The *requirements.txt* file is pretty simple; it contains just one line with `yattag`. But it can be more complex, and include multiple libraries and the versions of each library. The main reason to put the list of requirements in a file is to be able to store and automate the creation of the virtualenv.

Once we have *__main__.py* and *requirements.txt* we can build a virtualenv and store it in our directory with the following long command line:

```
$ docker run --rm \                                        ❶
  -v "$PWD:/tmp" \                                         ❷
  openwhisk/python3action bash -c \                        ❸
  "cd tmp && \                                             ❹
  virtualenv virtualenv && \                               ❺
  source virtualenv/bin/activate && \                      ❻
  pip install -r requirements.txt"                         ❼
```

❶ Start a "transient" Docker image.

❷ Mount the current directory to be accessible as */tmp* inside the Docker image.

❸ Launch the Docker image of the Python action.

❹ Go where our current directory is mounted.

❺ Build the virtualenv.

❻ Activate the virtualenv.

❼ Install the missing libraries.

After running the command, you will find a new directory in your current one—*virtualenv*—including `yattag` and all the other requirements that may depend on it.

Now it is time to zip the image, deploy it, and execute it:

```
$ zip -q -r yattag.zip __main__.py virtualenv                          ❶
$ wsk action update python/yattag yattag.zip \
   --web true --kind python:3                                          ❷
ok: updated action python/yattag
$ curl $(wsk action get python/yattag --url | tail -1)                 ❸
<html>                                                                 ❹
  <body>
    <h1>Hello world</h1>
  </body>
</html>
```

❶ Collect the action and its virtualenv.

❷ Deploy the action as a web action.

❸ Invoke the action as a web action with `curl`.

❹ The HTML output of the action as generated by `yattag`.

Using the OpenWhisk REST API

In earlier chapters, we saw how the OpenWhisk API allows us to invoke other actions, activate triggers, and execute asynchronous calls in JavaScript. OpenWhisk, however, is a multilanguage environment, so you may need APIs for other programming languages.

We are using Python in this chapter, but you could use Ruby, Go, Swift, PHP, etc. To support all these environments, OpenWhisk provides a generic REST API in *OpenAPI* (formerly Swagger) format. This API can be used by any programming language as soon as you have a library to use HTTP and JSON.

Let's look at how to use this API on the command line (with `curl`). After that, we'll see how to use it in Python.

Like all the REST APIs, you access the various functionalities using URLs as entry points. The generic format of the entry point is

```
https://{APIHOST}/api/v1/namespaces/{NAMESPACE}/{ENTITY}/...
```

where the names in curly braces are placeholders for specific values the user can specify. The *{APIHOST}* is where your OpenWhisk installation is running. If you are using,

for example, the IBM Cloud, it will be something like `openwhisk.eu-gb.bluemix.net`, but you may also have some URL specific to your installation if you chose to install it on your servers.

If you recall the discussion in Chapter 3, you may remember that the namespace is a name that you are assigned and is used as a common prefix for a set of OpenWhisk entities. The REST API places all the operations under the *`{NAMESPACE}`* because all the operations that refer to the *`{ENTITY}`* are restricted to entities present only in that namespace. After the namespace, you can specify the entity you want to operate with. Currently, the available entities are actions, triggers, rules, packages, and activations.

If you omit the *`{NAMESPACE}`*/*`{ENTITY}`* part it shows the available namespaces.

After the entity name, there are specific parameters for each entity. You can operate on each entity with HTTP methods like `GET`, `POST`, `PUT`, and others by providing a JSON payload that is different for each entity.

Let's see some of the possible combinations of methods and paths. We will not cover in detail every possible operation; instead, I'll show with examples only those I think are essential for practical use.

The CLI works by implementing the API and offers a verbose mode allowing you to see the actual URLs and JSON used. So, if you are in doubt about how to invoke certain operations using the REST API, you can always run the equivalent operations in the CLI and use the switch `-v` to see what is happening "under the hood."

Authentication

All the API operations are protected with HTTP Basic authentication. This means every request needs an authentication header with a value constructed by:

- Concatenating the username and password separated by a `:`
- Encoding the result in base64

Knowing how to build an authentication header is essential when you want to construct an appropriate HTTP request with a generic HTTP library. However, more advanced libraries often can generate this header for you.

In the IBM Cloud you may need to use something else in the authentication header, like the IAM bearer token.

Connecting to the API with curl

Now that we know how to format the URL we can try the first connection using the command-line tool `curl` with connection credentials available through the CLI.

If you type **`wsk property get`** you get output like this:

```
whisk auth                XXX:YYY
whisk API host            openwhisk.eu-gb.bluemix.net
whisk API version         v1
whisk namespace           learning_openwhisk
whisk CLI version         2018-09-14T17:43:44.288+0000
whisk API build           2018-09-20T14:24:50Z
whisk API build number    whisk-build-10300
```

where *XXX:YYY* is a long string encoding the username and password (it has been simplified in the example).

We have enough information to connect using the `curl` command. However, the command line and the URL would be very long—let's try to simplify them with environment variables.

Here is how to extract credentials from `wsk` and put in environment variables:

```
APIHOST=$(wsk property get --apihost | awk '{print $4}')                    ❶
APIVERSION=$(wsk property get --apiversion | awk '{ print $4}')             ❷
AUTH=$(wsk property get --auth | awk '{ print $3 }')                        ❸
NAMESPACE=$(wsk property get --namespace | awk '{ print $3}')               ❹
URL="https://$APIHOST/api/$APIVERSION/namespaces/$NAMESPACE"                ❺
```

❶ Extract the API host.

❷ Extract the version of the API.

❸ Extract authentication information.

❹ Extract the current namespace.

❺ The base URL to connect to.

With those variables we can easily invoke the list of packages. For example:

```
$ curl -s \                                                                 ❶
  -u $AUTH \                                                                ❷
  $URL/packages \                                                           ❸
  | jq .[1]                                                                 ❹
{
  "name": "python",
  "binding": false,
  "publish": false,
  "annotations": [],
```

```
  "version": "0.0.2",
  "updated": 1538230458075,
  "namespace": "openwhisk@example.com_dev"
}
```

❶ We use the `curl` command in `silent` (`-s`) mode to avoid unnecessary messages.

❷ We specify the username and password (`-u`).

❸ The URL was constructed using the variables we set before, adding the entity `packages`.

❹ Since the output is a long JSON array we use only the first one with this `jq` expression.

Alternatively, we can get a list of all the packages, picking only the name, with:

```
$  curl -u $AUTH -s $URL/packages | jq .[].name
"pattern"
"python"
"apidemo"
"basics"
"contact"
"contactdb"
```

When you use the command line, the command `jq` is handy since it lets you easily manipulate the JSON using its expression language. The subject is outside the scope of this book, but if you want to learn more visit the jq website.

We do not go into more detail here because we are more interested in using Python than `curl`. However, the ability to use the command line is invaluable when you need to debug.

Using the OpenWhisk REST API in Python

In Python, the most common library to perform HTTP requests is (not surprisingly) the `requests` library; it is not part of the standard library but is present in the set offered by the runtime. This library allows us to perform the same invocations that we did before using the command line. However, we need a few parameters we obtained before using `wsk` in the CLI. The same parameters are available in OpenWhisk, through the environment variables starting with the prefix `__OW_`. We can see those variables with this simple script:

```
import os
def main(args):
```

```
    m =  os.environ                                          ❶
    return { x:m[x]
             for x in m.keys()                               ❷
             if x.startswith("__OW_") }                      ❸
```

❶ Get the environment.

❷ Iterate on all the environment variables.

❸ Filter the environment variables starting with `__OW_`.

Using this script we can see that the same values we saw on the command line are also available here and are readily usable by Python scripts:

```
$ wsk action update python/environ environ.py
ok: updated action python/environ
wsk action invoke python/environ -r
{
    "__OW_ACTION_NAME": "/openwhisk@example.com_dev/python/environ",
    "__OW_ACTIVATION_ID": "81f048886f684f9eb048886f68cf9e46",
    "__OW_API_HOST": "https://eu-de.functions.cloud.ibm.com:443",
    "__OW_API_KEY": "XXXX:YYYY",
    "__OW_DEADLINE": "1538771271529",
    "__OW_NAMESPACE": "openwhisk@example.com_dev"
}
```

Now let's construct an action that can perform the same invocation we saw before listing the available packages. This time we will create a web request and produce a result in HTML.

To do this, we need to follow these steps:

1. Import the request library.
2. Get authentication credentials from the `API_KEY`.
3. Open a connection to the `API_HOST`.
4. Execute the request.
5. Wait for the result of the request and then parse it.
6. Process the result to return the final result.

Let's translate this plan into Python:

```
import os

def url(operation):
    return "%s/api/v1/namespaces/%s/%s" % (
        os.environ["__OW_API_HOST"],
        os.environ["__OW_NAMESPACE"],
```

```
            operation
        )
```

This code builds the REST URL we saw before using Python string formatting.

Let's see how we extract the authentication information:

```
def auth():
    up = os.environ['__OW_API_KEY'].split(":")
    return (up[0], up[1])
```

This code is slightly more interesting. In the API key, the username and password are concatenated and separated with `:`, while the `requests` library expects a tuple of the username and password. We need to separate the two components, splitting the string into two.

We are now ready to perform the request:

```
def whisk_get(operation):
    return requests.get(
         url=url(operation),
         auth=auth())
```

We merely construct a request object, passing the extracted URL and authentication information. This call executes the HTTP request and returns the result as a response object.

We know that the result is a JSON string that we can parse and process, as follows:

```
import json

def main(args):
    res = whisk_get("packages")                    ❶
    js = json.loads(res.text)                      ❷
    pkgs = [x["name"] for x in js]                 ❸
    return { "packages": pkgs }                    ❹
```

❶ Execute a `get` on the OpenWhisk URL to get the packages.

❷ Parse the JSON in a Python object.

❸ Extract the package name from a list of objects.

❹ Return the result.

Let's deploy the action and check the result:

```
$ wsk action update python/restpkgs restpkgs.py --kind python:3
ok: updated action python/restpkgs
$ wsk action invoke python/restpkgs -r
{
    "packages": [
        "python",
```

```
            "pattern",
            "apidemo",
            "basics",
            "contact",
            "contactdb"
        ]
    }
```

We got the same results as the `curl` examples in Python with an action running in OpenWhisk.

Invocations, Activations, and Triggers in Python

When you develop OpenWhisk applications, the more important APIs that we'll look at in detail are:

- Action invocations, either blocking or nonblocking
- Trigger invocations
- Activation retrieval

Blocking Action Invocation

Action invocation is probably the most important and frequently used function in the OpenWhisk API.

Action invocation has the following requirements:

- You must specify the action name in the URL.
- It receives additional information (the payload) that you want to submit.
- It can be either blocking or not blocking.
- It can return or not return the result of the invoked action.

An action invocation has the following URL format:

```
https://{APIHOST}/api/v1/namespaces/{namespace}/actions/[{packageName}/]\
{actionName}
```

You append to the URL as a parameter `blocking=1` if you want it to be blocking and `result=1` if you want to retrieve the result. Then, you need to send the data for the invocation in JSON format with a `POST` method; the returned data is in JSON format.

Now let's construct a `whisk_invoke` function that satisfies those requirements. First, let's define the parameters. You need to pass additional data, in the form of a Python object that can be serialized in JSON. Second, you can specify if you want the action invocation to be blocking and if you want the result of the invoked action. Since a

blocking action returning the result is the most common case, the default values of those parameters are `true`. The function has this signature:

```
def whisk_invoke(action, args,  blocking=True, result=True)
```

You can now invoke `whisk_invoke(action,args)` if you want a blocking invocation that returns the results. You can use `whisk_invoke(action, args, False)` if you want a nonblocking action invocation where you retrieve the results later. When you specify an action invocation with `whisk_invoke(action, args, False, False)` you are not interested in the results, so you "fire and forget." There is also the (rare) case of `whisk_invoke(action,args, True, False)`, when you want to block waiting for the action to complete but you are not interested in the results.

Let's take a look at the implementation. We can reuse the functions `url` and `auth` from before. Here we use a `POST` request to pass the payload, which must be encoded in JSON. We then decode the result. The function looks like this:

```
def whisk_invoke(action, args,
                 blocking=True, result=True):
    invoke = "actions/%s?blocking=%d&result=%d" % (                    ❶
        action, blocking, result)
    resp = requests.post(                                              ❷
        url=url(invoke),
        auth=auth(),
        json=args                                                      ❸
    )
    return json.loads(resp.text)                                       ❹
```

❶ Construct the URL in the required format.

❷ Note the request now is a `POST`.

❸ Pass the `args` with `json` to encode in JSON.

❹ Decode the JSON result as a Python object.

Let's now see how this works in practice. We'll use as an example the action `sort` from the `/whisk.system/utils` package. Hence, we'll first `bind` the package in our namespace to be able to use it without needing different credentials. Here's the `main` function from *invokesort.py*. The function simply accepts a text string as an input, splits it into words, and then returns an ordered list:

```
def main(args):
    input = {"lines": args["text"].split(" ")}                         ❶
    res = whisk_invoke("utils/sort", input)                            ❷
    return res                                                         ❸
```

❶ Prepare input, splitting the parameter `text`.

❷ Invoke the action `utils/sort`.

❸ Return the result, as is.

Now let's bind the package we want to invoke, deploy our function, and invoke it to test and see the result:

```
$ wsk package bind /whisk.system/utils utils                          ❶
ok: created binding utils
$ wsk action update python/invokesort \                               ❷
  invokesort.py --kind python:3
ok: updated action python/invokesort
wsk action invoke python/invokesort \                                 ❸
   -p text "b a d a c" -r
{
    "length": 5,
    "lines": [                                                        ❹
        "a",
        "a",
        "b",
        "c",
        "d"
    ]
}
```

❶ Now we have a `utils/sort` action.

❷ Deployment.

❸ Invocation.

❹ The result is the input line split and sorted.

Nonblocking Trigger Invocation

Action invocations can be blocking or nonblocking, depending on whether the parameter `blocking` is set to `true` or `false`. You can also invoke an action indirectly through a trigger and a rule. Trigger invocation, however, is always nonblocking. The invocation of a trigger is very similar to the invocation of an action, except you do not have to specify if it is blocking or not, and you cannot use package names in a trigger name. To invoke a trigger, the format of the URL is:

```
https://{APIHOST}/api/v1/namespaces/{namespace}/triggers
```

Reusing the usual `url` and `auth` functions, we can write a function that invokes a trigger named `python-trigger` (yet to be created) as follows:

```
def whisk_trigger(trigger, args):
    invoke = url("triggers/%s" % trigger)                             ❶
```

```
    resp = requests.post(
        url=invoke,
        auth=auth(),
        json=args
    )
    return json.loads(resp.text)

def main(args):
    input = {"lines": args["text"].split(" ")}
    return whisk_trigger("python-trigger", input)                    ❷
```

❶ Invoke the `trigger` entity.

❷ Specify the `python-trigger` as a trigger.

Now we can test build a trigger and a rule, and then fire the trigger to enable the rule and activate the action:

```
$ wsk trigger create python-trigger
ok: created trigger python-trigger
wsk rule update python-trigger-sort  python-trigger utils/sort
ok: updated rule python-trigger-sort
$ wsk action update python/firetrigger firetrigger.py --kind python:3
ok: updated action python/firetrigger
$ wsk action invoke python/firetrigger -p text "b a d a c" -r
{
    "activationId": "8023462aaa5e41fba3462aaa5ea1fbae"
}
```

We invoke our REST service in a nonblocking way. As you saw in earlier chapters, if you deploy and invoke the action again without the `-b` flag, it returns instead an activation id.

In general, however, you do not want to return the activation id to the user. You want to store it somewhere and use it later to retrieve the results. To complete our example we'll assume we pass the invocation id to another action (`python/retrieve`) that retrieves the result as follows:

```
input = {"lines": args["text"].split(" ")}
aid = whisk_invoke("utils/sort", input, False)
return whisk_invoke("python/retrieve", aid)
```

We delegate the retrieval of the data associated with the activation to another action because trying to retrieve the result immediately almost always leads to it not being available yet. So, we have to wait a bit using another invocation.

In general, when you have to wait, you can also use the `sleep` function.

Retrieving the Result of an Invocation

Now let's see how to retrieve the data associated with an activation id using the REST API. The URL has this format:

```
https://{APIHOST}/api/v1/namespaces/{namespace}/activations/{activationId}
```

Using `curl` you can see the format of the answer returned by the activation:

```
$ source init.sh
$ curl -u $AUTH -s $URL/activations/8023462aaa5e41fba3462aaa5ea1fbae| jq .
{
  "duration": 5,
  "name": "sort",
  "subject": "openwhisk@example.com",
  "activationId": "0e45f77a1cde47e385f77a1cded7e33d",
  "publish": false,
  "version": "0.0.92",
  "response": {
    "result": {
      "lines": [
        "a",
        "a",
        "b",
        "c",
        "d"
      ],
      "length": 5
    },
    "success": true,
    "status": "success"
  },
  "end": 1539110403714,
  "logs": [
    "2018-10-09T18:40:03.713748725Z stdout: sort input msg: \
       {\"lines\":[\"b\",\"a\",\"d\",\"a\",\"c\"]}",
    "2018-10-09T18:40:03.713814088Z stdout: sort before: b,a,d,a,c",
    "2018-10-09T18:40:03.713821123Z stdout: sort after: a,a,b,c,d"
  ],
  "start": 1539110403709,
  "namespace": "openwhisk@example.com_dev"
}
```

As you can see, the value returned is in the field `response.result`.

So, we can write an action receiving an activation id and execute the invocation. The `main` function is, as expected, just:

```
def main(args):
  id = args.get("activationId")                                    ❶
  if id:
    return whisk_activation(id)['response']['result']              ❷
  return {"error": "missing activationId"}
```

❶ Nonblocking action invocation.

❷ Retrieve the result of the invocation and extract the result.

Now we have all the pieces to make a round trip as follows (reusing the `whisk_invoke` and `whisk_trigger` functions we already wrote):

```
def main(args):
    input = {"lines": args["text"].split(" ")}
    aid = whisk_trigger("python-trigger", input)
    res =  whisk_invoke("python/retrieve", aid)
    return res
```

We show the results, deploying *retrieve.py* and *roundtrip.py* and testing it:

```
$ wsk action update python/retrieve retrieve.py --kind python:3
ok: updated action python/retrieve
$ wsk action update python/roundtrip roundtrip.py --kind python:3
ok: updated action python/roundtrip
$ wsk action invoke python/roundtrip -p text "b a d a c" -r
{
    "lines": [
        "b",
        "a",
        "d",
        "a",
        "c"
    ]
}
```

Testing Python Actions

You already know how to write and run actions in OpenWhisk. Now you need to learn how to debug and test those actions. You can, of course, debug your actions by deploying them and then using logs to inspect their behavior when they do not work correctly. But this is not a very effective way of debugging, since it involves a lot of repetitive and time-consuming work.

As we saw earlier for JavaScript, a better way to debug Python actions is by executing them locally, applying unit tests to the various parts of your code before deploying it in the runtime.

Hence, I am now going to show the essential steps required to perform a useful test:

- How to recreate the environment of the runtime locally
- How to write unit tests, using examples
- How to execute locally actions that invoke the REST API
- How to "mock" remote requests to get predictable results

We'll reuse the *rstpkgs.py* code (the script that shows a list of our packages) to explore how to test in Python.

Recreating the Python Runtime Environment Locally

The first step to test your actions locally is to recreate the runtime environment. You should install the same version of the interpreter available in the runtime and create the same dependencies. As shown in "What's in the Python Runtime?" on page 152, you can quickly inspect the packages in use and create a *requirements3.txt* file specifying the exact versions of the packages you need. Once you have that, you can create a virtualenv for local use, installing the exact versions that are in the runtime.

Let's execute at the terminal the initialization of a test environment for Python 3. I assume you have deployed the `python/requirements3` action to retrieve the list of the requirements.

```
$ curl -s $(wsk action get python/requirements3 --url \
 | tail -1) >requirements3.txt
$ head requirements3.txt
zope.interface==4.5.0
wheel==0.31.1
Werkzeug==0.14.1
w3lib==1.19.0
virtualenv==15.1.0
Twisted==17.1.0
six==1.11.0
simplejson==3.10.0
setuptools==40.4.3
service-identity==17.0.0
```

We can now use this file to create a virtual environment with the right libraries. We'll also add a "mocking" library for HTTP, called `httpretty`, to use in our tests:

```
$ virtualenv virtualenv  --python python3
Running virtualenv with interpreter /usr/local/bin/python3
New python executable in virtualenv/bin/python3.6
Also creating executable in virtualenv/bin/python
Installing setuptools, pip, wheel...done.
$ source virtualenv/bin/activate
(virtualenv) $ pip install -r requirements3.txt
... omitted ...
```

```
$ pip install httpretty
Installing collected packages: httpretty
Successfully installed httpretty-0.9.5
```

You can find documentation about the `httpretty` library at HTTPretty.

Unit Test Examples

There are many ways to write unit tests. One of the easiest ways to write tests in Python uses *examples*, a technique similar to using snapshots in JavaScript. One feature of Python that everyone loves is the interactive interpreter, because it allows you to execute Python code on the fly. We will use it here.

Using examples to test means that:

- You execute your code in the interpreter, to test it and verify the results.
- When you get the correct results, you save the examples, embedding them in your program.
- You run the test by executing the commands again at the command line and compare if the results are still the same.

So, let's start adding tests to all the functions in the *restpkgs.py* file. We'll begin with function `auth` shown in "Using the OpenWhisk REST API in Python" on page 165, and try to test it in the Python interpreter. The function is pretty simple—it only reads the environment variable `__OW_API_KEY` and splits its value into two:

```
$ python                                                              ❶
Python 3.6.5 (default, Jun 17 2018, 12:13:06)
[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information
>>> import restpkgs as r, os                                          ❷
>>> os.environ['__OW_API_KEY']="USER:PASS"                            ❸
>>> r.auth()                                                          ❹
('USER', 'PASS')                                                      ❺
```

❶ Invoke the Python interpreter at the command line.

❷ Import the action as a module, with a short name for convenience.

❸ Explicitly set the environment variable read by the action to a known value.

❹ Invoke the function.

❺ This is the result, as expected.

Now we can transform the log of the terminal into a test very quickly: we just need to add the transcript in the documentation string, or docstring, for our function. A documentation string in Python is just a string constant placed at the beginning of the body of our function code. It serves no purpose other than generating documentation or, as in our case, testing. Here's the docstring of the `auth` function, including the test:

```
def auth():
    """ Extract OpenWhisk authentication keys
    >>> import restpkgs as r, os
    >>> os.environ["__OW_API_KEY"]="USER:PASS"
    >>> r.auth()
    ('USER', 'PASS')
    """
    up = os.environ['__OW_API_KEY'].split(":")
    return (up[0], up[1]
```

Yep, it's nothing more than the transcript of our interpreter session. It is useful as documentation, but it can also be used as an executable (and hence repeatable) test!

In the standard Python library there is a module called `doctest` that allows you to collect the output of an interpreter session, embed it in the source code as a documentation string, and then use it as an executable test, repeating the session and comparing the results. You can read more at the doctest library.

Let's see how to execute the test. In Python, it is possible to make any module executable locally on the command line. Each script has a variable `__name__` that is set to the value `__main__` when the script is invoked directly by the interpreter as the main entry point. We can leverage this feature to run our tests, embedded as documentation. All you need to do is make each module executable and invoke the function `doctest.testmod()` as follows:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Now, you can execute the module to run the tests, as follows. If you run it and don't get any output at all, this is good news. Silence means your code passed the tests. But let's see what happens when there is an error. Let's change the value assigned to the environment variable from `USER:PASS` to `USERNAME:PASSWORD`:

```
> python restpkgs.py
**********************************************************************
File "restpkgs.py", line 24, in __main__.auth
Failed example:
    r.auth()
```

```
Expected:
    ('USER', 'PASS')
Got:
    ('USERNAME', 'PASSWORD')
**********************************************************************
1 items had failures:
   1 of   3 in __main__.auth
***Test Failed*** 1 failures.
```

If we now change the embedded listing, replacing `('USER', 'PASS')` with `0, 'PASSWORD'`), again we don't get an answer, as expected.

Invoking the OpenWhisk API Locally

So far, we've seen how to run a Python action locally, creating an environment similar to the runtime and adding the ability to execute it as a standalone program. However, in OpenWhisk we also can invoke other actions or fire triggers using the API, as discussed in detail in "Using the OpenWhisk REST API in Python" on page 165.

Invoking the API inside OpenWhisk requires environment variables that are set by the runtime before executing our action. Our action will not run locally because those environment variables are missing. Obviously, the solution is to set those variables locally. Luckily, the values we need are also the values the `wsk` CLI uses, and are stored in a local file (the file *.wskprops* in the home directory). Names and values are not the same as environment variables, but they are close enough that they are pretty easy to convert.

So, to be able to run tests using the OpenWhisk API we need the following function, `load_props`, which extracts the OpenWhisk parameters and sets the local environment variables using the `wsk` command:

```
def load_props():
  import os, os.path
  with open(os.path.expanduser("~/.wskprops"), "r") as f:                    ❶
    for line in f.readlines():
        [k, v] = line.strip().split("=")                                     ❷
        if k == "AUTH": os.environ["__OW_API_KEY"] = v                       ❸
        if k == "NAMESPACE": os.environ["__OW_NAMESPACE"] = v                ❹
        if k == "APIHOST": os.environ["__OW_API_HOST"] = "https://%s:443" % v ❺
```

❶ Open the *.wskprops* file and read it line by line.

❷ Separate each `k=v` string into two values.

❸ Set the `__OW_API_KEY` environment variable from the `AUTH` value.

❹ Set `__OW_NAMESPACE` from the `NAMESPACE` value.

❺ Set `__OW_API_HOST` in a format slightly different from the `APIHOST` value.

For example, using this function, we can test the `whisk_get` function that relies on a remote call to the OpenWhisk API:

```
def whisk_get(operation):
    """ Execute GET requests on the OpenWhisk API
    >>> import restpkgs as r                                    ❶
    >>> r.whisk_init()                                         ❷
    >>> re = r.whisk_get("packages")                           ❸
    >>> re
    <Response [200]>                                           ❹
    >>> type(re.json())
    <class 'list'>                                             ❺
    """
    return requests.get(
         url=url(operation),
         auth=auth())
```

❶ Import the module under test.

❷ Here we initialized the environment, so it is now possible to invoke the OpenWhisk API.

❸ Get the list of packages.

❹ Verify that the response is correct.

❺ Verify that we got a list as expected.

This test is a bit vague, but we can't really be any more specific because the list of packages depends on the current configuration of your environment. We can just verify that we can connect to the server and that the answer is a JSON list. To be able to be more precise we need to be able to impose the result returned, and we can do this with *mocking*.

Mocking Requests

As you saw in Chapter 6, mocking is a technique that allows you to replace a real-world response—one that involves interaction with external systems—with a simulated one for testing purposes. This feature is critical for testing OpenWhisk actions that have complex interactions with other actions.

Mocking is a generic concept, and there are advanced libraries you can use to do it. However, in the OpenWhisk environment most of the time you need to be able to mock the result of HTTP calls, replacing specific requests to real-world systems with values determined at test time.

Let's see how to do that when testing the code for the `main` function of our *restpkgs.py* action:

```
def main(args):
    res = whisk_get("packages")
    js = json.loads(res.text)
    pkgs = [x["name"] for x in js]
    return { "packages": pkgs }
```

The problem here is that `whisk_get("packages")` returns different values according to the state of the system, and it is difficult to run this test without forcing a specific system configuration. For this reason, we need to force (or mock) the answer for the HTTP request to a known value.

For this purpose, we use the library `httpretty` we included in the local environment for testing. Let's see the test showing the interaction in the Python interpreter:

```
>>> import restpkgs as r, httpretty as h, json                          ❶
>>> r.whisk_init()                                                     ❷
>>> h.enable()                                                         ❸
>>> resp = json.dumps([{"name":"first"}, {"name":"second"}])           ❹
>>> h.register_uri(h.GET, r.url("packages"), body=resp)                ❺
>>> r.main({})                                                         ❻
{'packages': ['first', 'second']}                                      ❼
>>> h.disable()                                                        ❽
```

❶ Import the modules we are going to use.

❷ Initialize the OpenWhisk environment variables.

❸ Enable the mocking library to replace responses.

❹ The response we want to force from `whisk_get("packages")`.

❺ The mocking request that will force our response.

❻ Here, we invoke the `main` function that will, in turn, do an HTTP request whose result will be replaced by our mock.

❼ Given the mocking data this is the expected result.

❽ We can now disable the mocking library to avoid side effects.

Now, to create an embedded test we only need to collect the interaction at the terminal and insert it in a docstring for the `main` function.

Summary

In this chapter, you learned how to write OpenWhisk actions using Python. First, you learned about the peculiarities of the OpenWhisk Python runtime and the ways to write and package Python applications that may include third-party libraries using virtual environments.

Then we explored how to connect to the OpenWhisk API, using the REST interface and writing code to invoke other actions, fire triggers, and retrieve results.

Finally, we went through testing actions using the Python interpreter, embedding tests in code and using mocking libraries.

CHAPTER 8

Using CouchDB with OpenWhisk

In the first part of the book, when we were developing our contact form we actually used a database to store data. At the time, we didn't go into much detail about how the database works, because the focus of the first part was understanding OpenWhisk.

But at this point, we need to know more about how to use a database since data storage is a key component of every nontrivial application.

Or course, we can't possibly cover every single database out there—we have to choose one. OpenWhisk offers a package that integrates with the Cloudant and CouchDB databases, NoSQL databases based on JSON that fit the JSON-based paradigm of OpenWhisk pretty well, so the choice is obvious.

Cloudant is an IBM product, a cloud-based NoSQL database offered as a software service in the cloud. Cloudant is based on the open source Apache project CouchDB, which can be deployed into on-premises deployments of OpenWhisk. OpenWhisk is not bound to Cloudant, since you can also use CouchDB. Everything in this chapter applies to both Cloudant and CouchDB. For simplicity, in the rest of the chapter I'll talk about CouchDB, but it also applies to Cloudant.

CouchDB is a good match for OpenWhisk because it is JSON based, it is schemaless, and it is scalable.

Let's discuss those points. First, all the actions in OpenWhisk talk to each other and exchange JSON objects. Hence, JSON is the natural format for managing data. The obvious requirement in similar environments is a database that can persist JSON objects. This is exactly what CouchDB does, as it stores JSON objects as "documents."

JSON objects are not stored exactly "as is": to be able to index and retrieve data, CouchDB adds additional fields that act as identifiers in the database.

Because CouchDB can store JSON objects there is no need to provide a "schema," as in a SQL database. You do not have to declare fields; CouchDB can store any format of JSON object, eventually adding some additional fields for its own use.

As with any database, scalability is important. Cloudant is a commercial product, distributed and optimized for heavy workloads like those generated by mobile applications and websites with a lot of traffic. It is offered by IBM as a platform, and it is available in the IBM Cloud.

CouchDB is an open source product offered by the Apache Software Foundation. It can be clustered, allowing it to scale, but you have to build a cluster in your deployment if you need it. You can learn more by reading the full documentation or the free guide.

The source code for the examples related to this chapter is available in the GitHub repository.

How to Query CouchDB

With any database, you have to know how to query it for it to be of any use. CouchDB does not use the familiar SQL language. Instead, it has two different ways to query: the first one is a JSON-based query language called *Mango* and the second is a MapReduce approach using JavaScript functions embedded in the database. You embed JavaScript functions in CouchDB using specially crafted JSON files called *design documents*.

Let's first discuss the query language. It was inspired by the query language of another widely used NoSQL database, MongoDB (hence the name). Using this language, you express queries in a format that can be considered as the JSON equivalent of SQL statements. To give you an idea of how this works, let's look at an example using SQL (a language many people are familiar with):

```
SELECT name FROM person
```

Using Mango, you can express a similar query in this format:

```
{
  "selector": {
     "type": "person"
```

```
    },
    "fields: ["name"]
}
```

We cover more of the details in "Querying CouchDB" on page 191.

For now, let's look at how JavaScript uses MapReduce. MapReduce is essentially a pattern for distributed computation. It was popularized by Google, which used it to implement large-scale distributed queries for a search engine.

This pattern works in two steps. In the first step, *mapping*, you invoke a `map` function on each item of the data. Each mapping produces values that are further classified using keys.

After you run the mapping step, then you perform a *reduction* on the mapped data, generally to aggregate the results. This step works by invoking a `reduce` function on the output of the first step, this time grouped in batches, where the grouping is performed using keys.

CouchDB uses MapReduce both for creating derivate representations of data stored in the database (called *views*) and to calculate aggregate functions (sums, averages, etc.).

CouchDB implements MapReduce using function, embedded in the database that are written in JavaScript (although they can also be written in other languages, like Erlang).

The two approaches are both useful. Using the query language you can easily extract data, which covers the more common use cases. Using MapReduce is similar to adding stored procedures to the database; the database performs most of the work for you and you just need to extract the results.

Exploring CouchDB on the Command Line

To get a feeling for how CouchDB works before delving into the details of the interaction with OpenWhisk, I am going to explore the interaction with it using the command-line interface, with the help of common tools like `curl` and `jq`.

You can register in the IBM Cloud and get a free instance of Cloudant for development and test purposes. In Figure 2-5 I showed all of the steps required to get access and retrieve a username and password. That information is enough to access the database from the command line since the username is also used as the hostname.

To play with Cloudant at the command line, you need to set a couple of environment variables (`CLOUDANT_USER` and `CLOUDANT_PASS`) with the values retrieved from the IBM Cloud and set a couple of other variables (`URL` and `AUTH`) for convenience, as follows:

```
$ export CLOUDANT_USER="<username-from-ibm-cloud>"          ❶
$ export CLOUDANT_PASS="<password-from-ibm-cloud>"          ❷
$ export URL="https://$CLOUDANT_USER.cloudant.com"          ❸
$ export AUTH="$CLOUDANT_USER:$CLOUDANT_PASS"               ❹
```

❶ Replace the username with the actual value

❷ Replace the password with the actual value

❸ Set the `URL` to access Cloudant

❹ Set the `AUTH` as the username/password to access

Using just those two variables and `curl` is enough to get access to Cloudant. For example, invoking the entry point `$URL`, you get the version:

```
$ curl -u $AUTH $URL | jq .
{
  "couchdb": "Welcome",
  "version": "2.1.1",
  "vendor": {
    "name": "IBM Cloudant",
    "version": "7137",
    "variant": "paas"
  },
  "features": [
    "geo",
    "scheduler",
    "iam"
  ]
}
```

How CouchDB works

Next, look at the basic operations. You need to understand these since the package operations you will see and use are a wrapper on top of this REST API.

In particular, let's learn the basic CRUD operations, most notably how to:

- Create a database
- Insert data in the database
- Retrieve data from the database
- Update data from the database
- Delete data from the database

Those are just the basic features of the database. There are many others I am going to explore later.

Creating Database

The username in the IBM cloud is also the hostname of a Cloudant instance. You can also have a local instance of CouchDB (and set your `$URL` variable to point to it).

In a CouchDB instance, there can be multiple databases. A database is a logical partition of data. In CouchDB, a database is identified by a URL. For example, everything under `$URL/demodb` belongs to the `demodb` database.

```
To create such a database, do the following:

$  curl -X PUT -u $AUTH $URL/demodb                    ❶
{"ok":true}                                            ❷
```

❶ Note the method `PUT` used to create the database

❷ CouchDB tells you the database was created

You can check whether the database was actually created with the special URL `_all_dbs`, which lists all the available databases:

```
$  curl -u $AUTH $URL/_all_dbs                         ❶
["contactdb","demodb","patterndb"]                     ❷
```

❶ Without parameters the request is a `GET`

❷ You see the new database and others created before

You may have noticed the `_all_db` URL. In general, in CouchDB both URLs and field names starting with _ are special and are generally managed by the database for special purposes.

Create

Now let's explore how to execute the basic CRUD (create, retrieve, update, and delete) operations of any database using the JSON-based paradigm of CouchDB.

CouchDB stores in its database JSON objects that are called *documents*. A document is any JSON object (but not an array or a single value) with two important properties added: `_id` and `_rev`. `_id` is a unique identifier (either generated or provided by the user) used to identify the document within the database, while `_rev` is another identifier, managed entirely by CouchDB and used to distinguish among different versions of the same document.

In CouchDB, creating a new object is just a matter of performing a `PUT` HTTP request against the URL where we want to store the object. The URL is the name of the data-

base followed by the id of the document. The database returns a confirmation message with the id and the revision:

```
$  curl -u $AUTH -X PUT $URL/demodb/msciab \                    ❶
    -d '{"name": "Michele", "age": 50}'                         ❷
{"ok":true,                                                     ❸
  "id":"msciab",                                                ❹
  "rev":"1-4cbf3dded8477e95b5079eda6038c22d"}                   ❺
```

❶ Request to store a JSON document.

❷ The data is the JSON object you want to store.

❸ Confirm everything is OK.

❹ The `_id` of the object.

❺ The `_rev` (revision ID) of the object.

A revision id is an incremental number followed by a special value calculated by CouchDB. Technically speaking it is the MD5 hash of the transport representation, but this really does not matter for practical use. What matters is that you have to use it for updates. It is there to avoid multiple clients modifying an object at the same time, to be sure only one succeeds and the others fail.

Retrieve

Now let's verify what happened. We can try to retrieve the document at the given URL using a `GET` request. As you might expect, we receive the document with the special properties `_id` and `_rev`:

```
$ curl -su $AUTH -X GET $URL/demodb/msciab \                    ❶
  | jq .
{
  "_id": "msciab",                                              ❷
  "_rev": "1-4cbf3dded8477e95b5079eda6038c22d",
  "name": "Michele",                                            ❸
  "age": 50
}
```

❶ Request the document with id `msciab` in database `demodb`.

❷ CouchDB added `_id` and `_rev`.

❸ The other fields of the document.

Update

Updating works in a similar way to creating, except you have to provide the revision id of the current document to be sure you are actually updating the document you read and not an older version that may have been modified.

You can see this by trying again to `PUT` the same object under the same URL—you get an error:

```
$ curl -u $AUTH -X PUT $URL/demodb/msciab  \                          ❶
  -d '{"name": "Michele Sciabarra", "age": 50}'
{"error":"conflict",                                                  ❷
 "reason":"Document update conflict."}                                ❸
```

❶ Try to use `PUT` again with the same URL as before.

❷ You get an error.

❸ There is a conflict because there is no revision provided.

To fix this, add the revision in the URL. You just need to append `?rev=`*`<id>`* , where *`<id>`* is the revision id returned in the preceding call:

```
$ curl -u $AUTH -X PUT \
  "$URL/demodb/msciab?rev=1-4cbf3dded8477e95b5079eda6038c22d" \       ❶
   -d '{ "name": "Michele Sciabarra", "age": 50}'
{"ok":true,
 "id":"msciab",
 "rev":"2-f8885327dee4d24fb8edff31e7683962"}                          ❷
```

❶ The old revision id.

❷ The new revision id.

You can now do a simple check to verify that the document was actually updated:

```
$ curl -su $AUTH -X GET $URL/demodb/msciab | jq .
{
  "_id": "msciab",
  "_rev": "2-f8885327dee4d24fb8edff31e7683962",
  "name": "Michele Sciabarra",                                        ❶
  "age": 50
}
```

❶ Change the name as requested.

Delete

The last operation in the classic CRUD set of operations is delete. In CouchDB you use the `DELETE` HTTP method followed by the revision number:

```
$  curl -su $AUTH -X DELETE                                          ❶
   "$URL/demodb/msciab?rev=2-f8885327dee4d24fb8edff31e7683962"
{"ok":true,"id":"msciab",
 "rev":"3-022254165a2971263cca5b2b5f108d58"}
```

❶ The `DELETE` operation.

Let's verify if the document is now deleted:

```
$ curl -su $AUTH -X GET $URL/demodb/msciab | jq .
{
  "error": "not_found",
  "reason": "deleted"
}
```

It is.

Attachments

In a relational database, you have BLOB fields holding binary data. In CouchDB you have attachments. This means you can upload a file in binary format (not JSON) and retrieve it as is. CouchDB works a bit like a writable web server. But when you upload an attachment, it is always part of a document. The URL format for attachments inside a server is:

```
/<database>/<document>/<attachment>
```

So you can have `/demodb/openwhisk/logo.png` but not `/demodb/logo.png` nor `/logo.png`. For example, let's create an attachment containing the OpenWhisk logo. We first create a document containing the BLOB, then we upload it. We use the `PUT` method and the revision id because uploading an attachment is actually an update. It is also important to upload the data as a binary file and to specify the `Content-Type` header. Note that for attachments the type is not just `application/json` but varies according to the actual type of the attachment:

```
$ curl -u $AUTH -XPUT \                                             ❶
   $URL/demodb/openwhisk \
  -d '{"width":800,"height":400}'
{"ok":true,
 "id":"openwhisk",
 "rev":"1-29ee2a1bdb910c06e36b432922a3bd42"}
 $ curl -u $AUTH -XPUT \                                            ❷
  "$URL/demodb/openwhisk/logo.png\                                  ❸
  ?rev=1-29ee2a1bdb910c06e36b432922a3bd42"\                         ❹
   --data-binary @openwhisk.png \                                   ❺
   -H "Content-Type: image/png"                                     ❻
{"ok":true,
 "id":"openwhisk",
 "rev":"2-c4603cf28267f4cd8284936f37a2e211"}
```

❶ Create the containing document.

❷ Create the attachment.

❸ The URL of the attachment, placed under a document in a database.

❹ The revision id, since we are updating a document.

❺ Here we specify that we want to upload the file *openwhisk.png*.

❻ The content type of the attachment, an image.

If we check the document we can see that there is an attachment. However, the attachment body is not shown:

```
$ curl -su $AUTH $URL/demodb/openwhisk | jq .
{
  "_id": "openwhisk",
  "_rev": "2-c4603cf28267f4cd8284936f37a2e211",
  "width": 800,
  "height": 400,
  "mime": "image/png",
  "_attachments": {
    "logo.png": {
      "content_type": "image/png",
      "revpos": 2,
      "digest": "md5-lKbApb8v7JnR5EDeh0/8zA==",
      "length": 27191,
      "stub": true
    }
  }
}
```

You can now see the attachment using a browser. To do so, you need to construct an appropriate URL that also includes authentication information.

In our examples, we used two variables: `$AUTH` containing the authentication information and the `$URL` to the Cloudant instance. The `$URL` has the format `https://<database-server>`. You need to put the authentication information in the URL.

An authenticated URL has the format `https://<user>:<pass>@<database-server>`. We can easily construct the authenticated URL variable `$AURL` by removing the prefix and including the authentication information:

```
URL1="${URL##https://}"                                    ❶
AURL=https://$AUTH@$URL1                                   ❷
export AURL
```

❶ Remove the protocol from the URL.

❷ Add it again with the authentication information.

Now we have a variable we can use to access the attachment with a browser. How you open the browser depends on the system. You can simply echo the `$AURL` and copy and paste it in your browser, or open the browser from the command line. On macOS you can open the URL in Google Chrome with:

```
$ open -a "Google Chrome" \
 $AURL/demodb/openwhisk/logo.png
```

The result is shown in Figure 8-1.

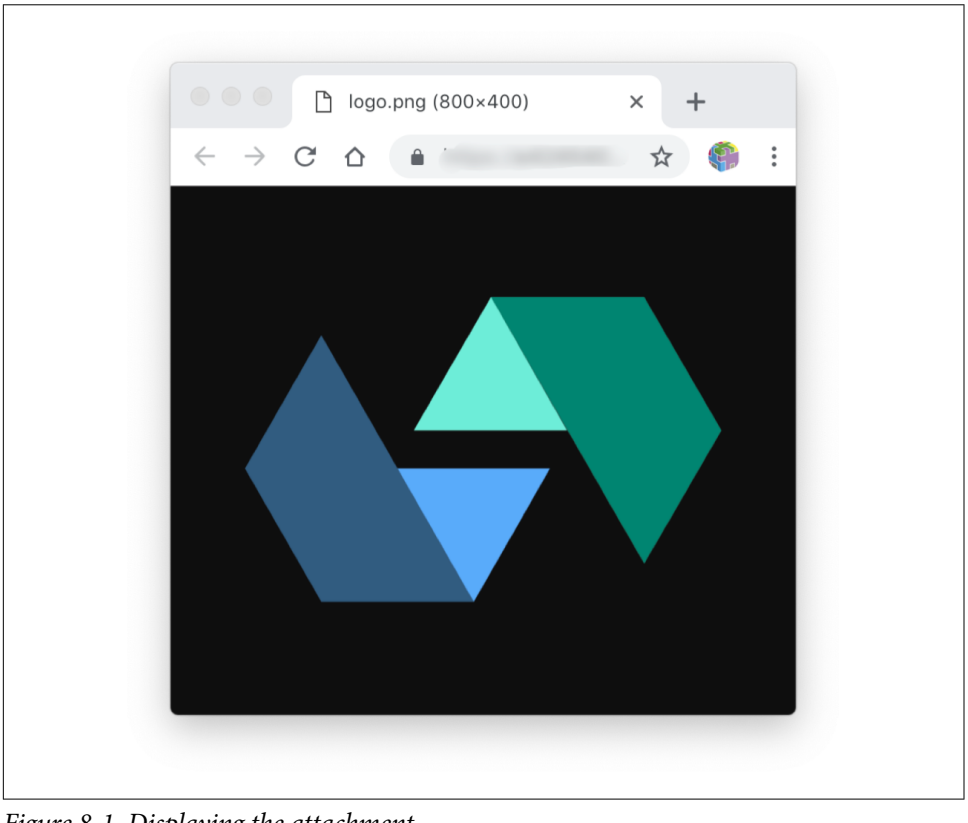


Figure 8-1. Displaying the attachment

In other systems, like Windows, you may want to use the command `start` to open the URL in the default browser. In Linux, you generally directly invoke a browser like Firefox with the URL on the command line.

To be able to display an image, a browser needs to know the MIME type. The image is displayed correctly because the database knows its MIME type, but generating URLs accessing the database directly is not recommended since it would expose security information. There are ways to make attachments public, but I recommend using a web action and exporting images instead (explained in the next chapter) for this purpose.

Querying CouchDB

Now that you know how to actually create data, let's see how to do queries in CouchDB using the Mango query language. We start by loading some sample data in the database (see Tables 8-1 and 8-2). In the GitHub repository for the book, you can find the data in JSON format and a Makefile to import it.

Table 8-1. Persons

_id	type	name	dob
mike	person	Michele	1968-07-09
miri	person	Mirella	1966-10-25

Table 8-2. Computers

_id	type	name	brand	owner	memory
mac	computer	MacBookPro	Apple	mike	16
pc	computer	Pavilion	HP	miri	8
pc2	computer	m15	Alienware	mike	32

Searching the Database

You can query your database using `<database>/_find`. In the simplest form, a query is just a JSON object with a field named `"selector"` whose value can be an empty object. The following JSON object just selects all the documents in the repository:

```
{ "selector": {} }
```

But notice that not all the documents available are returned. There is an implicit limit of 25 documents, and the server returns a "bookmark," an opaque identifier letting you get the next batch of documents.

In the body of the selector, you can include some values restricting the documents you retrieve. In the simplest case just specify a field and a value, and you will get only the documents with that field and that value. Here is an example:

```
$ curl -su $AUTH -X POST $URL/demodb/_find \                    ❶
  -H "Content-Type: application/json" \                         ❷
  -d '{"selector":{ "type": "person" }}' \                      ❸
```

```
  | jq .
{
  "docs": [
    {
      "_id": "mike",
      "_rev": "1-7e2e7dcf6ae46e373bf9836fec2cbb04",
      "dob": "1968-07-09T23:00:00.000Z",
      "name": "Michele",
      "type": "person"
    },
    {
      "_id": "miri",
      "_rev": "1-34500a0cd43eec3ba6f1d5f44943b365",
      "dob": "1966-10-25T23:00:00.000Z",
      "name": "Mirella",
      "type": "person"
    }
  ],
  "bookmark":
  "g1AAAAA4eJzLYWBgYMpgSmH..."                                    ❹
   "warning": \                                                   ❺
  "no matching index found, create an index to optimize query time"
}
```

❶ Invoke the URL to query the database using `POST`.

❷ The content type is required.

❸ This selector extracts all the documents of type `person`.

❹ This bookmark helps to paginate things (explained later).

❺ Note this warning, because there is no index for the field `type`.

Indexes

In the last example, there is a warning telling you that the field you queried for is not indexed. Indeed, to get better performances when you query the database you need to create an index. An index is, in its simplest form, a list of field names that are likely to be searched for. Hence, the database needs to keep track of these field names in a special way to be able to find more quickly documents containing those fields.

You can describe the indexes using a JSON file called *index.json*, like this:

```
{
  "index": {
    "fields": [
      {
        "type": "asc"
      }
```

```
      ]
    }
}
```

This JSON file declares that you want to index the field `type`, in ascending order (alternatively you can use `desc`). You can put the index description in a file called *index.json* and deploy it with a `POST` to a database with the URL `_index` as follows:

```
$ curl -su $AUTH $URL/demodb/_index \                                    ❶
-X POST -H "Content-Type: application/json" \                            ❷
-d @index.json                                                           ❸
{
  "result": "created",
  "id": "_design/d0ad4137f94d6923bfcff603fe3984cf1e21af1d",
  "name": "d0ad4137f94d6923bfcff603fe3984cf1e21af1d"
}
```

❶ The special URL to create an index.

❷ Note that when you `POST` you need to specify the content type.

❸ Include the content of the file *index.json*.

Now, if you run the previous query again, the warning will no longer be there.

Fields

In the query object, you can specify the fields you want to extract. In the preceding example you saw that by default CouchDB returns all fields, but since that can be a lot, you generally don't want to do this. You can filter the fields you need using the `fields` attribute by specifying an array with the field names you want to extract. For example, let's try the following query that extracts just the field `name` from all the documents in our database:

```
$ curl -su $AUTH -X POST $URL/demodb/_find  \
  -H "Content-Type: application/json" \
  -d '{"selector":{ },                                                   ❶
       "fields": ["name"] }' \                                           ❷
  | jq .docs                                                            ❸
[
  {
    "name": "MacBookPro"
  },
  {
    "name": "Michele"
  },
  {
    "name": "Mirella"
  },
  {
```

```
      "name": "Pavilion"
    },
    {
      "name": "m15"
    }
  ]
```

❶ Select all the documents.

❷ Extract only the field `name`.

❸ Show only the results (property `docs`).

Pagination Support

CouchDB supports pagination with many options available in the query object. Using those options it is easy to navigate the various documents. In the query object, you can specify the number of documents returned with the `limit` property. As already mentioned, this property defaults to 25, so you never get more than 25 results, unless you specify a different value. There is another property you may need: `skip`. This lets you skip the given number of documents. Let's try an example, restricting the view to three documents and skipping one:

```
$ curl -su $AUTH -X POST $URL/demodb/_find \
  -H "Content-Type: application/json"
  -d '{"selector":{ },        \
       "fields": ["name"],    \
       "limit": 3,            \                    ❶
       "skip": 1}'            \                    ❷
  | jq .
 {
  "docs": [
    {
      "name": "Michele"
    },
    {
      "name": "Mirella"
    },
    {
      "name": "Pavilion"
    }
  ],
  "bookmark":
  "g1AAAAA0eJzLYWBgYMpgSmHgKy5JLC"                   ❸
}
```

❶ Limit the number of the documents returned to three.

❷ Skip one document at the beginning.

❸ Take note of this identifier as we are going to use it again.

As you can see, compared with the previous result, where there were five documents; you see only three of them and the first one has been skipped.

Bookmark Feature

In the previous example, we paged using `limit` and `skip`. With many databases you can do only this, using a `skip` parameter to reach the current page. However, CouchDB does better by offering the more efficient "bookmark" feature. Each result also returns an identifier, called the bookmark, which can be used to ask for the next batch of the search without needing to keep track of where the cursor is (usually the tricky part when doing pagination). For example, if we want to continue the search of the previous batch, we can just use the bookmark returned at the end of the query. Let's try it:

```
$ curl -su $AUTH -X POST $URL/demodb/_find \
  -H "Content-Type: application/json" \
  -d '{"selector":{ },
       "fields": ["name"],
       "limit": 3,
       "bookmark":"g1AAAAA0eJzLYWBgYMpgSmHgKy5JLC"}' \                    ❶
  | jq .
{
  "docs": [
    {
      "name": "m15"
    }
  ],
  "bookmark": "g1AAAAA2eJzLYWBgYMpgSmHgKy5JLC"
}
```

❶ Specify the bookmark to continue the search.

As you can see, we are now getting the last document out of the five we found in the first search. We limited the search to three results, but only one was missing, and it is now shown.

Selectors

This section goes into more detail on the syntax of selectors, which are required to perform more complex queries than the simple one you have just seen.

A selector is a JSON object with a mandatory key `selector` whose value contains the query options. As you have already seen, you can define an empty value (`{}`) to select everything, or specify a fixed key-value like `{"type":"person"}` to select all the documents having that key with the specified value.

Actually, you can put in multiple values. Here's see an example with two values:

```
$ curl -su $AUTH -X POST $URL/demodb/_find \
  -H "Content-Type: application/json"
  -d '{"selector":{
       "type":"computer",                                    ❶
       "owner":"mike"}                                       ❷
   }' | jq .docs
[
  {
    "_id": "mac",
    "_rev": "1-6fb66eb7faba0b9ebe4bfdbb0da31616",
    "name": "MacBookPro",
    "brand": "Apple",
    "owner": "mike",
    "type": "computer",
    "memory": 16
  },
  {
    "_id": "pc2",
    "_rev": "1-c1381146da5ef5da5215cd78a02cb3ce",
    "name": "m15",
    "brand": "Alienware",
    "owner": "mike",
    "type": "computer",
    "memory": 32
  }
]
```

In the database, there are five records; selecting `type=computer` you restrict the search to the three available while selecting `owner=mike` you find only those two owned by him.

Now, the example you have just see is really an abbreviation for a more complex syntax. The inner constraint `{"type": "computer"}` is really a short form for a clause `{"type": {"$eq": "computer"}}`, so it actually defines a constraint requiring that `type` is `$eq` to `compute`. implicitly defines an equality relationship.

Furthermore, when you declare multiple clauses, they are actually conditions implicitly connected by a `$and` relationship. The condition of the preceding example fully expanded is hence:

```
{ "$and": [
    { "type": {"$eq", "computer" }
    },
    { "owner": {"$eq": "mike" }
    }
  ]
}
```

Each clause applies an operator to the specified field and the given value; then it calculates the `and` of all the intermediate results.

If you already know the SQL, you could write a query more or less equivalent as:

```
SELECT *
FROM EVERYTHING
WHERE type  = "computer"
AND   owner = "mike"
```

Note that generally in SQL data are stored in different tables, while in CouchDB there is not a table concept. So in the example, I put a hypothetic `EVERYTHING` table, a fictitious table containing all the fields for all the documents.

Operators

So far you have seen only the `$eq` and the `$and` operator. There are actually many others, as shown in Table 8-3.

Table 8-3. Some Frequently Used Query Operators

operator	meaning
$and	if all selectors in array matches
$or	if one selector in array matches
$not	if the given selector does not match
$all	if an array contains all the elements of the given array
$type	check for the JavaScript type
$ne / $ne	compare JSONs for equality / inequality
$lt /$lte	compare JSON for less / less or equal
$gt, $gte	compare JSON for greater / greater or equal
$exist	check if a field exists
$in, $nin	check if a field is / is not in a given array
$regexp	check if a field matches a regular expression

As you can guess, you can do much more complex queries with those operators. For example, I want to search for all the computers either having more than 16 GB of memory, or its brand is either "Apple" or "Dell."

You need to use different operators. To select the memory size, you need to use the operator `$gt` to filter everything greater than 16, while you use the `$in` operator for filtering for values belonging to a given set. Last but not least, conditions are now connected by the `$or` operator.

Expressing this query in JSON looks like this:

```
{ "selector": {
    "$or": [
      "memory":
        {"$gt": 16},
      "brand": {
        "$in": ["Apple", "Dell"] }
    ]
  }
}
```

I save now the query in a file `query.json` and run it as follows:

```
$ curl -su $AUTH -X POST $URL/demodb/_find \
   -H "Content-Type: application/json"  \
   -d "$(cat query.json)" \                                    ❶
   | jq .docs
[
  {
    "_id": "mac",
    "_rev": "1-6fb66eb7faba0b9ebe4bfdbb0da31616",
    "name": "MacBookPro",
    "brand": "Apple",
    "owner": "mike",
    "type": "computer",
    "memory": 16
  },
  {
    "_id": "pc2",
    "_rev": "1-c1381146da5ef5da5215cd78a02cb3ce",
    "name": "m15",
    "brand": "Alienware",
    "owner": "mike",
    "type": "computer",
    "memory": 32
  }
]
```

❶ Loading the query from the `json` file

You can see that I find a computer `Apple` (that has 16GB RAM) from the first clause, and another that does not belong to the given brands, but it has 32 GB of RAM so it matches the second clause.

CouchDB Design Documents

In addition to JSON-based queries, you can write queries for CouchDB using JavaScript and MapReduce functions. Databases usually have the ability to store recurring operations you can call later. Those are called *stored procedures*, and CouchDB uses JavaScript for this purpose. CouchDB MapReduce functions extend the database with functions stored in so-called *design documents*. A design document is a JSON object

that can be stored in CouchDB and looks like an ordinary document (with an `_id` and a `_rev`), except it embeds some JavaScript functions used by the database to perform various tasks.

A design document can actually define a number of different functions:

View
: Functions performing the MapReduce we discussed in "How to Query CouchDB" on page 182.

Validation
: Functions invoked on updates to validate values.

Show
: Functions useful for producing a rendering of data in the database in various formats (not only JSON but also HTML, XML, SVG, etc.).

List
: Functions that can transform the results of view functions, to generate renderings like those produced by show functions.

Update
: Functions that perform server-side logic for creating or modifying documents.

Filter
: CouchDB has a "log" of all the changes that can be listened for; these functions can manipulate this log, for example, to reduce the notifications produced to a more interesting subset.

In the context of OpenWhisk, the more useful functions are the view and validation functions. Those are the cases we'll discuss in detail. View functions allow advanced ways to query the database. Validation functions allow writing in the database documents that satisfy certain requirements.

For examples of other functions, refer to the CouchDB documentation.

Creating a Design Document

A design document that stores the functions you just saw has a specific format. It is a JSON file where some of the fields are JavaScript functions; you manage it like a normal document except you have to use the special path `_design`.

A design document is formatted like the following example. Note that we only look at view and validation functions here. The markers in angle brackets, like *<name>*, are not part of the JSON but must be replaced by the user:

```
{
  "_id": "_design/<name>",                                    ❶
  "views": {                                                  ❷
    "<view-name>": {                                          ❸
       "map": "<map-function>",                               ❹
       "reduce": "<reduce-function>"                          ❺
    }
  },
  "validate_doc_update": "<validation-function>"              ❻
}
```

❶ Name of the design document.

❷ Views are under this key.

❸ Each entry introduces a view with a given name.

❹ Key to define a `map` function.

❺ Key to define a `reduce` function.

A design document is still a JSON file, not a JavaScript one. Hence, JavaScript functions are stored in the design document as simple strings. Furthermore, in JSON a string cannot be multiline, so all the newlines must be represented as `\n`.

Design documents are treated like other documents, so they are uploaded and updated with `PUT`, using `_id` and `_rev`. What distinguishes these files from others is the URL, which is *<database>*`/_design/`*<name>*. Once you have created the document, there are other URLs you can use to invoke the embedded functions. For example, to invoke the view we can use *<database>*`/_design/`*<document>*`/_view/`*<name>*. We'll discuss this in the next section.

View Functions

In CouchDB, originally there were only view functions to query the database. You can now also use the Mango query language, but views are still the more efficient and powerful querying mechanism.

A *view* is a transformation of the data contained in a database, calculated by a couple of functions; the first is called `map` and the second `reduce`. Note that the second is

optional (we'll discuss it later). Reductions are useful to perform aggregations, but many useful tasks can be performed using only `map`.

Extracting Data with map Functions

A `map` function is a JavaScript function that receives a document and processes it, returning a derivation of it. CouchDB invokes a `map` function on all the documents of the database. For each document, it can produce a single derivation, multiple derivations, or none. You do not return the derivation; instead, you use the function `emit` to produce it. Each derivation includes a key and a value. Keys are optional (they can be null) but are important since they are used for a number of purposes, such as selecting the output of a view, ordering, and reduction.

Let's write a simple view, returning all the names in our database, using the type as the key. The `map` function will receive each document and emit the `doc.type` as the key, and the `doc.name` as the value.

First we'll construct the design document:

```
{
 "_id" : "_design/sampleviews",                                    ❶
 "views" : {
   "names":  {                                                     ❷
     "map":
     "function(doc) { emit(doc.type, doc.name) }"                  ❸
    }
  }
}
```

❶ The design document name.

❷ The view name.

❸ The `map` function.

Now we can publish it using the variables `$URL` and `$AUTH`:

```
$ curl -su $AUTH -X PUT \
  $URL/demodb/_design/sampleviews \
  -d @sampleviews.json | jq .
{
  "ok": true,
  "id": "_design/sampleviews",
  "rev": "49-4d0b58f7906bd1e6d57fa5ad6c14ba58"
}
```

Finally, we test it, invoking the view `names` using a URL of the form *<database>*/_design/*<document>*/_view/*<name>*:

```
$ curl -su $AUTH \
  $URL/demodb/_design/sampleviews/_view/names
{"total_rows":5,"offset":0,"rows":[
{"id":"mac","key":"computer","value":"MacBookPro"},
{"id":"pc","key":"computer","value":"Pavillion"},
{"id":"pc2","key":"computer","value":"m15"},
{"id":"mike","key":"person","value":"Michele"},
{"id":"miri","key":"person","value":"Mirella"}
]}
```

The result gives a "view" of the names in the database, in a format that is easy to process: you get all the names, organized by document type.

The URL used for invoking the view accepts many parameters. You can use the parameter `key` to receive only the records with a given type. So, to get all the names of the persons involved you can use:

```
$ VIEW=$URL/demodb/_design/sampleviews
$ curl -su $AUTH \
  "$VIEW/_view/names?key=\"person\"" \                    ❶
   jq -r ".rows[] | .value"                              ❷
"Michele"
"Mirella"
```

❶ Restrict the result to the output with this key.

❷ Filter the values from the output.

The key must be a true JSON value, so it has to be `"person"` with quotes, not `person` without. This is why we had to add the backslashes, to include quotes within quotes. The entire query had to be quoted because the `?` is a shell metacharacter and it would be expanded without the quotes.

We used `jq` to extract the values from the result on the command line. In general, you extract the values in your user code, or you can ask CouchDB to do it for you using a list function (which we do not discuss here).

You do not have to emit all the documents in the database. A common practice is just to emit the documents that match some criteria. For example, if you want to select only persons in the database, just emit documents with `doc.type == "person"`, adding this entry to the file *sampleviews.json*:

```
"persons": {
  "map":
  "function(doc){ if(doc.type == 'person') emit(doc._id, doc.name) }"
},
```

The result is:

```
$ curl -su $AUTH "$VIEW/_view/persons"
{"total_rows":2,"offset":0,"rows":[
{"id":"mike","key":"mike","value":"Michele"},
{"id":"miri","key":"miri","value":"Mirella"}
]}
```

Implementing a Join with map Functions

CouchDB is not a relational database; it is document-oriented and does not implement relations as a primitive concept. Joining two tables, an operation very simple with a traditional relational database, is not a native function in CouchDB. Instead, you replace `JOIN` with some techniques to get this result using `map` functions.

In general, when you want scalability and to process big data, you should rely less on normalizing the database (a classic operation for a relational database). In a scalable database, it is common to duplicate data and avoid normalization of the database.

Let's demonstrate how to implement a join in CouchDB with an example. In our demo database there are computers, and for each computer there is a field called `owner` that refers to its owner. We want to implement a query to join each computer with the person that owns it. In SQL terms this is roughly equivalent to a query like this:

```
SELECT *
FROM PERSON
INNER JOIN COMPUTER
ON COMPUTER.ID = PERSON.OWNER
```

Let's do the same in CouchDB. Emitting multiple documents that contain the ids of the records to join is the recommended way to implement joins with views. You will leverage the following features:

- You do not have to emit a single document in your view functions, so you will emit more than one: one for the main record and one for each joined record.
- If a view has a field `_id`, you can use the option `include_docs` to expand it and include all the data from the referred-to document.
- Returning more documents of the view under the same key helps to retrieve all the data for the join.

Use these steps to perform a join:

1. In your `map` function, first emit the `_id` of the main record as the value, under some key.

2. Then emit the `_id` of each of the joined records as the value, under the same key as the main record.
3. Invoke the view with the option `include_docs=true`.
4. Select all the records you want to use the key.

For example, let's assume you want to join a document with type `computer` with documents with type `person` over the field `owner` of a computer record. This is a simplification, in general, you can emit multiple records when you join, but the technique is the same. You can do this by adding a function like this:

```
{
 "views": {
   "join": {
     "map": "function(doc) {                                    ❶
        if(doc.type == 'computer') {                            ❷
          emit(doc.owner, { _id: doc._id} );                    ❸
          emit(doc.owner, { _id: doc.owner })                   ❹
         }
      }"
    }
   }
}
```

❶ Simplified JSON for readability; must be a single file with quoted newlines.

❷ Select computers.

❸ Emit a value with the id of the computer.

❹ Emit a value with the id of the person.

Now publish the design document as discussed before, and let's look at the result:

```
$ curl -u $AUTH "$VIEW/_view/join?\
include_docs=true\
&key=\"miri\"" | jq .
{
  "total_rows": 6,
  "offset": 4,
  "rows": [
    {
      "id": "pc",
      "key": "miri",
      "value": {
        "_id": "pc"
      },
      "doc": {
        "_id": "pc",
        "_rev": "1-f381af8cac6c3cf040fa9a87bb5b05dd",
```

```
          "name": "Pavillion",
          "brand": "HP",
          "owner": "miri",
          "type": "computer",
          "memory": 8
        }
      },
      {
        "id": "pc",
        "key": "miri",
        "value": {
          "_id": "miri"
        },
        "doc": {
          "_id": "miri",
          "_rev": "1-34500a0cd43eec3ba6f1d5f44943b365",
          "dob": "1966-10-25T23:00:00.000Z",
          "name": "Mirella",
          "type": "person"
        }
      }
    ]
}
```

As you can see, the result is actually spread across two records. The actual values are in the `doc` field. Your user code expects multiple records and gathers the values accordingly.

CouchDB guarantees the results will be ordered by key. In this case, we just filtered the result using a single key, but you can use multiple keys. Also, when you expect results with a variable number of included documents, you can use the `startkey` and `endkey` parameters for filtering. I recommend checking the CouchDB documentation documentation for more details, as it includes a more detailed example with multiple records.

Joining with a Single Document

The solution for joining documents in the previous section works, but it requires more processing of the output.

In the frequent case of a document joined with another single document, you can actually get only one document with all the data as output. You can do this by:

- Placing all the properties of the main document as the value,
- Using the id of the joined document as the `_id`
- Expanding it with `include_docs`

For example, let's try filtering the result by computer id and retrieving all the properties of the computer and the owner:

```
{
 "views": {
    "join2": {
      "map": "function(doc) {                                    ❶
        if(doc.type == 'computer') {
          emit(doc.id,
           { _id: doc.owner,                                     ❷
             name: doc.name,                                     ❸
             brand: doc.brand,
             memory: doc.memory })
         }
      }"
    }
  }
}
```

❶ Simplified for readability; should be on a single line.

❷ Use the id of the joined document.

❸ Add as values all the properties of the document.

Now let's try to select the computer with the id `mac` and see its owner:

```
$ curl -su $AUTH "$VIEW/_view/join2\
  ?include_docs=true&key=\"mac\"" | jq .
{
  "total_rows": 3,
  "offset": 0,
  "rows": [
    {
      "id": "mac",
      "key": "mac",
      "value": {
        "_id": "mike",
        "name": "MacBookPro",
        "brand": "Apple",
        "memory": 16
      },
      "doc": {
        "_id": "mike",
        "_rev": "1-7e2e7dcf6ae46e373bf9836fec2cbb04",
        "dob": "1968-07-09T23:00:00.000Z",
        "name": "Michele",
        "type": "person"
      }
    }
  ]
}
```

The result is spread out, but you have all the values of the two joined records.

Aggregations with reduce Functions

So far we've only discussed view functions with a `map` part. We know how those functions can implement the SQL equivalent of selects and joins. Now let's look at reduction functions, the equivalent in CouchDB of SQL aggregate functions. For example, let's consider the SQL statement:

```
SELECT COUNT(*)
FROM SOME_TABLE
```

How can we do this in CouchDB? The trick is to first map all the documents (to extract the information to aggregate), then invoke the `reduce` function on the mapped values to produce the final result. For example, if you want to count the number of documents, you just map each document into the value `1`, then sum all the values. In code, you can do something like the following:

```
"count": {
    "map" :
    "function(doc){ emit(doc.type, 1)}",                    ❶
    "reduce":
    "function(key, values) { return sum(values) }"          ❷
}
```

❶ For each document, emit a value of `1`.

❷ Sum all the values to get the count.

Let's check the result:

```
$ VIEW=$URL/demodb/_design/sampleviews
$ curl -su $AUTH $VIEW/_view/count
{"rows":[
 {"key":null,"value":5}
]}
```

This works, but it's not the most efficient way to perform this aggregation. There are three built-in aggregation functions that can do a more efficient job because they are written in the language of CouchDB (Erlang):

- `_count` counts the number of emitted values.
- `_sum` assumes the emitted values are numbers, then adds them up.
- `_stats` assumes the emitted values are numbers, then calculates some statistical values—the minimum, maximum, sum, count, and sum of squares.

For example, using those functions you can write a more efficient `ncomputer` view that returns the number of computers:

```
"ncomputer": {
  "map" : "function(doc) {                                    ❶
      if(doc.type == 'computer')
         emit(doc._id) }",
  "reduce": "_count"                                          ❷
},
```

❶ Simplified; must be on a single line.

❷ Using the built-in `_count`.

Now let's look at an example using the `_stat` function on all the documents. Add this view to your design document:

```
"stats": {
    "map" : "function(doc){ emit(doc._id, 1)}",
    "reduce": "_stats"
}
```

Let's see the result:

```
$ curl -su $AUTH $VIEW/_view/stats | jq .
{
  "rows": [
    {
      "key": null,
      "value": {
        "sum": 5,
        "count": 5,
        "min": 1,
        "max": 1,
        "sumsqr": 5
      }
    }
  ]
}
```

Validation Functions

CouchDB can validate a document when you create or modify it. Validation is performed by adding a validation function to the design document. This validation function either completes without any result if everything is okay or can you notify of errors by throwing exceptions. To create this function, you insert the key `validate_doc_update` in the design document, using the function as a value. If everything is okay the result is ignored. If there are exceptions, the document won't be created or changed and the invoker will be notified of the error.

Let's go over an example of this type of validation function—validating contacts. For a contact that has only a name and an email address, we want to be sure they are both

specified. Furthermore, we want email addresses in a certain format, so we'll check them against a regular expression.

Validation functions are invoked for every update in the database, but we want to validate only for contacts, so the first step is filtering for `doc.type == "contact"`. Then we'll apply some checks to the properties of the document.

The function is complex enough that we should write the code in a separate *validation.js* file:

```
function (doc, old, ctx) {
    if(doc.type && doc.type == "contact") {                    ❶
        if(!doc.name) {                                        ❷
            throw({forbidden: "name required"})
        }
        if(!doc.email) {                                       ❸
            throw({forbidden: "email required"})
        }
        var re = /\S+@\S+\.\S+/;                               ❹
        if(!re.test(doc.email)) {                              ❺
            throw({forbidden: "not an email"})
        }
    }
}
```

❶ Filter the validation only for contacts.

❷ Check that there is a name.

❸ Check that there is an email address.

❹ A regular expression to validate the email address.

❺ Validate the email address against the regular expression.

We can now post the function to CouchDB. This time we want to solve the problem of encoding a JavaScript function into JSON. We need a *validate.json* document in the format:

```
{
  "_id": "_design/validate",
  "validate_doc_update": "<validate-function-here>"
}
```

Since the function is not a trivial one-liner we store it in a separate *validate.js* file, then ask the `jq` utility for help. Before posting to CouchDB, we generate *validate.json* from *validate.js* as follows:

```
$ jq -n --rawfile file validate.js   \                         ❶
 '{ "_id" : "_design/validate",    \
```

```
    "validate_doc_update":$$file}' \                                    ❷
        >validate.json                                                  ❸
```

❶ Provide our JSON as a variable to `jq`.

❷ Expand the variable inside a template.

❸ Save the result.

With a validation function in place, the effect is that now some document updates may fail. Let's try it:

```
$ curl -u $AUTH $URL/demodb/test-mike \
  -X PUT -d '{"type":"contact"}'
{"error":"forbidden","reason":"name required"}                          ❶
$ curl -u $AUTH $URL/demodb/test-mike \
  -X PUT -d '{"type":"contact","name": "Mike", \
  "email": "mike@home"}'                                                ❷
{"error":"forbidden","reason":"not an email"}
$ curl -u $AUTH $URL/demodb/test-mike
  -X PUT -d '{"type":"contact","name": "Mike", \
   "email": "m@s.c"}'                                                   ❸
{"ok":true,"id":"test-mike",
"rev":"3-00673090ce0564363fc27285873f4bc4"}
```

❶ Error, no name.

❷ Error, wrong format for the email address.

❸ Everything is okay now.

Using the Cloudant Package

Now that you know more about CouchDB and Cloudant, you probably want to start using it with OpenWhisk. The examples in this chapter all refer to Cloudant running in the IBM Cloud, but everything discussed here also applies to a local installation of OpenWhisk, provided you also have your own instance of CouchDB.

The OpenWhisk package `/whisk.system/cloudant` provides a high-level interface to Cloudant/CouchDB. While it is possible to directly use the database using the REST API, it is easier with this package.

Note that you need to:

- Authenticate in Cloudant/CouchDB to use the package.
- Select a database to work with it.

- Authenticate in OpenWhisk to be able to invoke an action.

You can do all of this in a single step by binding the `cloudant` package and specifying the database as a package parameter. But note that:

- When an action invokes a database action, it uses its own OpenWhisk authentication information to invoke the other action.
- The package contains authentication information to access the database.
- The package automatically selects which database to use.

To see how this works, let's create a package binding to access one specific database in one instance of Cloudant. The binding must set the username, password, and current database in the package variables. The following assumes that you've already set the `CLOUDANT_USER` and `CLOUDANT_PASS` environment variables to the values you have retrieved in the IBM Cloud:

```
$ wsk package bind /whisk.system/cloudant demodb \                    ❶
 -p username "$(CLOUDANT_USER)" \                                     ❷
 -p password "$(CLOUDANT_PASS)" \
 -p host "$(CLOUDANT_USER).cloudant.com" \                            ❸
 -p dbname demodb                                                     ❹
```

❶ Bind `demodb` to the `cloudant` system package.

❷ Provide the username and password.

❸ The hostname in Cloudant is just the username followed by a standard suffix.

❹ Select the database that you want to use with this package.

The binding is now available—you can access an instance, but you still need to create a database. Let's do this using the bound package itself. This is equivalent to the `curl` command we used to create the database:

```
$ wsk action invoke demodb/create-database -r                         ❶
{
    "ok": true                                                        ❷
}
```

❶ Create the database.

❷ It worked.

You are now ready to work with your shiny new Cloudant database.

It is worth mentioning that in the IBM Cloud you can get an automatic binding for a provisioned Cloudant database with the command `ibmcloud fn package refresh`. However, the name is automatically generated and the binding does not select a fixed database.

CRUD Actions in the Cloudant Package

The standard OpenWhisk `cloudant` package offers actions to do CRUD operations, and they are simpler to use than the CouchDB API. Let's see how to use it with a number of examples.

First, let's inspect the package itself to see what functions are available:

```
$ wsk action list demodb | awk '{ print $1}' | sort
/whisk.system/cloudant/delete-attachment
/whisk.system/cloudant/update-attachment
/whisk.system/cloudant/read-attachment
/whisk.system/cloudant/create-attachment
/whisk.system/cloudant/read-changes-feed
/whisk.system/cloudant/delete-query-index
/whisk.system/cloudant/delete-view
/whisk.system/cloudant/manage-bulk-documents
/whisk.system/cloudant/exec-query-view
/whisk.system/cloudant/exec-query-search
/whisk.system/cloudant/exec-query-find
/whisk.system/cloudant/list-query-indexes
/whisk.system/cloudant/create-query-index
/whisk.system/cloudant/list-design-documents
/whisk.system/cloudant/list-documents
/whisk.system/cloudant/delete-document
/whisk.system/cloudant/update-document
/whisk.system/cloudant/write
/whisk.system/cloudant/read-document
/whisk.system/cloudant/read
/whisk.system/cloudant/create-document
/whisk.system/cloudant/read-updates-feed
/whisk.system/cloudant/list-all-databases
/whisk.system/cloudant/delete-database
/whisk.system/cloudant/read-database
/whisk.system/cloudant/create-database
/whisk.system/cloudant/changes
```

For each action, you also can get information about the available parameters. For example, if you want to know which parameters `create-document` accepts you can use:

```
$ wsk action get demodb/create-document | tail +2 | jq .annotations
[
  {
    "key": "description",
```

```
      "value": "Create document in database"
    },
    {
      "key": "parameters",
      "value": [
        {
          "name": "dbname",
          "required": true
        },
        {
          "description": "The JSON document to insert",
          "name": "doc",
          "required": true
        },
        {
          "name": "params",
          "required": false
        }
      ]
    },
    {
      "key": "exec",
      "value": "nodejs:8"
    }
]
```

This action requires a database name with `dbname`, a document with `doc`, and additional parameters with `params`, which is optional. The parameter `dbname` is specified at the package binding level, so it is always present for each invocation of actions in the `demodb` package. The only required parameter for creating a document is thus `doc`. Let's try `create-document`, passing a simple document:

```
wsk action invoke demodb/create-document \
  -p doc '{"name":"Mike"}' -r
{
    "id": "9a2360ec53b50942fbae59b4476e1895",
    "ok": true,
    "rev": "1-4ad839e8dccd6221eb78f388ad6b9f98"
}
```

Note that the created document generates a new id; if you invoke it multiple times, you will get multiple records with the same content. Let's try to invoke it again and then list all the available documents in the database:

```
$ wsk action invoke demodb/create-document \
  -p doc '{"name":"Mike"}' -r
{
    "id": "7fdecd38345fa3487fa15f83a983b847",
    "ok": true,
    "rev": "1-4ad839e8dccd6221eb78f388ad6b9f98"
}
$ wsk action invoke demodb/list-documents -r
```

```
{
    "offset": 0,
    "rows": [
        {
            "id": "7fdecd38345fa3487fa15f83a983b847",
            "key": "7fdecd38345fa3487fa15f83a983b847",
            "value": {
                "rev": "1-4ad839e8dccd6221eb78f388ad6b9f98"
            }
        },
        {
            "id": "9a2360ec53b50942fbae59b4476e1895",
            "key": "9a2360ec53b50942fbae59b4476e1895",
            "value": {
                "rev": "1-4ad839e8dccd6221eb78f388ad6b9f98"
            }
        }
    ],
    "total_rows": 2
}
```

As expected, invoking `create-document` twice in an empty database results in two documents. If you want to delete the second document, inspecting the action `delete-document` you'll see that there are two required parameters: `docid` and `docrev`.

As you learned in "Delete" on page 187, you need a revision id to delete a document. Let's try it:

```
$ wsk action invoke demodb/delete-document \
 -p docid 9a2360ec53b50942fbae59b4476e1895 \
 -p docrev 1-4ad839e8dccd6221eb78f388ad6b9f98 -r
{
    "id": "9a2360ec53b50942fbae59b4476e1895",
    "ok": true,
    "rev": "2-984386b290c0601c61cdb83fd932442b"
}
$ wsk action invoke demodb/list-documents -r
{
    "offset": 0,
    "rows": [
        {
            "id": "7fdecd38345fa3487fa15f83a983b847",
            "key": "7fdecd38345fa3487fa15f83a983b847",
            "value": {
                "rev": "1-4ad839e8dccd6221eb78f388ad6b9f98"
            }
        }
    ],
    "total_rows": 1
}
```

If you instead use `update-document`, you will see it uses just a generic `doc` parameter; this is a Couch DB document, so you have to add the `_id` and `_rev` parameters as Couch DB prescribes. Hence, you have to read the entire document to get the `_id` and `_rev` fields, change something, then submit the document again to get the changes.

Let's start by reading the document:.

```
$ wsk action invoke demodb/read-document \
   -p docid 7fdecd38345fa3487fa15f83a983b847 -r \
  | tee doc.json                                          ❶
{
    "_id": "7fdecd38345fa3487fa15f83a983b847",
    "_rev": "1-4ad839e8dccd6221eb78f388ad6b9f98",
    "name": "Mike"
}
```

❶ This command saves the input in a file and also shows it in the terminal.

Now you can use the `jq` utility to change a field of the document and save it in another file:

```
$ jq '.name = "Michele Sciabarra"' \
  <doc.json | tee doc1.json
{
  "_id": "7fdecd38345fa3487fa15f83a983b847",
  "_rev": "1-4ad839e8dccd6221eb78f388ad6b9f98",
  "name": "Michele Sciabarra"
}
```

Finally, you can submit the updated document:

```
$ wsk action invoke demodb/update-document -r \
  -p doc "$(cat doc1.json)"                               ❶
{
    "id": "7fdecd38345fa3487fa15f83a983b847",
    "ok": true,
    "rev": "2-dc5a2d3358cefb0567df64086faa8696"
}
```

❶ Read the entire document from the file and then pass it as a parameter on the command line.

If you try to read the document again, you will see it has been updated.

Queries and Views with Packages

Of course, it is possible to search in the database using either the Mango query language or views. The action to invoke to execute a Mango query is `exec-query-find`, using the parameter `query`. This parameter must be a query in the format you saw in "Querying CouchDB" on page 191.

We won't repeat all the examples here. Instead, here's a simple query with `type` equaling `person` using the test data from before:

```
$ wsk action invoke demodb/exec-query-find \
  -p query '{"selector":{"type":"person"}}' -r
{
    "bookmark": "g1AAAAA4eJzLYWBgYMpgSmHgKy5JLC...",
    "docs": [
        {
            "_id": "mike",
            "_rev": "1-7e2e7dcf6ae46e373bf9836fec2cbb04",
            "dob": "1968-07-09T23:00:00.000Z",
            "name": "Michele",
            "type": "person"
        },
        {
            "_id": "miri",
            "_rev": "1-34500a0cd43eec3ba6f1d5f44943b365",
            "dob": "1966-10-25T23:00:00.000Z",
            "name": "Mirella",
            "type": "person"
        }
    ]
}
```

Now let's look at how to invoke one of the `view` functions defined in "Implementing a Join with map Functions" on page 203. In particular, we'll see how to invoke the view `join2`, passing a parameter to expand the document and filtering using a key.

The action to invoke is `exec-query-view`, and you have to specify the `docid` (the ID of the design document, without the `_design` prefix), `viewname` (the name of the view), and `params` (a JSON object with the additional parameters to pass):

```
$ wsk action invoke demodb/exec-query-view \
  -p docid sampleviews \
  -p viewname join2 \
  -p params '{
    "include_docs": true,
    "key": "pc"}'
  -r | jq .
{
    "offset": 1,
    "rows": [
        {
            "doc": {
                "_id": "miri",
                "_rev": "1-34500a0cd43eec3ba6f1d5f44943b365",
                "dob": "1966-10-25T23:00:00.000Z",
                "name": "Mirella",
                "type": "person"
            },
            "id": "pc",
```

```
            "key": "pc",
            "value": {
                "_id": "miri",
                "brand": "HP",
                "memory": 8,
                "name": "Pavillion"
            }
        }
    ],
    "total_rows": 3
}
```

Summary

In this chapter we learned about CouchDB (and its cloud-based brother, Cloudant), the NoSQL database that powers OpenWhisk and is packaged with all the installations.

First, we learned the basics of CouchDB: creating, updating, and deleting documents (actually, JSON objects).

Then we went through specific ways of querying CouchDB: either with the Mango query language or with `map` and `reduce` functions.

Finally, we explored advanced functions, like bookmarks, pagination, validation, joining, and design documents.

CHAPTER 9

An OpenWhisk Web Application in Python

In this chapter, we are going to develop a non-trivial example of an OpenWhisk Python application using CouchDB/Cloudant. The application we are going to develop is a database table editor. The goal is to demonstrate coding in Python, by creating a web user interface that interacts with databases in OpenWhisk. We also cover testing in depth.

For illustration purposes, the application is actually split into two implementations. First we'll build a basic application, with bare-bones functionality. Then we'll create a more advanced implementation, with more complex features.

The source code for the examples related to this chapter is available in the GitHub repository.

CRUD Application Architecture

In the basic application, we cover:

- Implementation and testing of database operations in Python
- Creation and testing of a simple HTML user interface
- Implementation and testing by "mocking" of the application control logic

The application, shown in Figure 9-1, has a simple structure. It uses only HTML tables and forms, and a bit of client-side JavaScript. We keep the application features to an absolute minimum here and instead focus on OpenWhisk coding techniques, avoiding too many implementation details. We follow the usual Model-View-

Controller pattern for a web application, splitting it into three modules: *model.py*, *view.py*, and *control.py*.

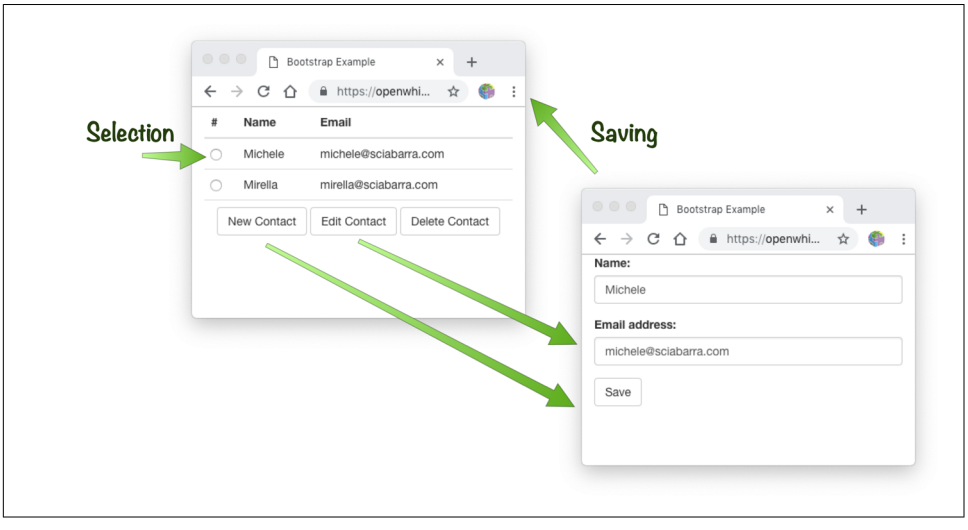


Figure 9-1. CRUD application user interface

When invoked, the application shows a list of the current contacts. Records just include a name and email address. You can add new contacts by clicking New Contact. You can update an existing contact by selecting it and clicking Edit Contact. Finally, selecting a contact and clicking Delete Contact, will delete the contact.

The application has two main states: the "table" and "form" states. The table state displays the records. Figure 9-2 shows how the application works as a state machine. You can go into the form state with the operations *new* and *edit*. If you perform the operation *delete*, you return to the table state. Finally, from the form state you can come back to the table state with the *save* operation.

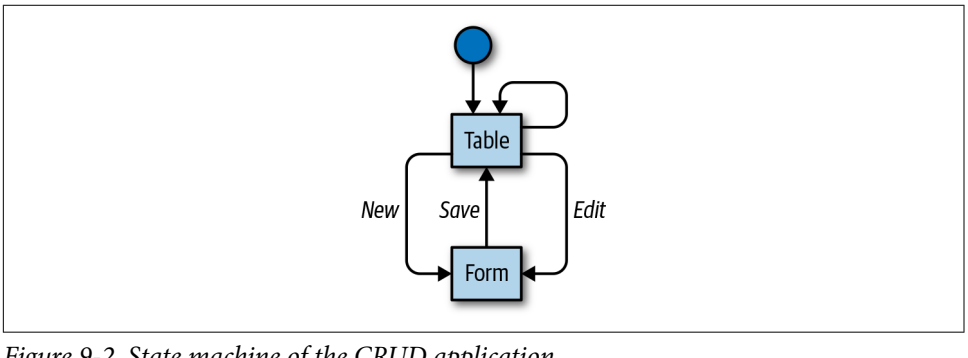


Figure 9-2. State machine of the CRUD application

Deploying the Action

Let's begin by deploying the `crud` action. While we don't get into the coding right now, I will show the beginning code here because we need to define an environment at deployment time.

The application is comprised of the following files, which we will explore in detail in the following sections:

- *model.py*, which interfaces with the database
- *view.py*, which generates the HTML user interface
- *control.py*, which handles user interaction
- *rest.py*, the interface to the OpenWhisk REST API
- *__main__.py*, the entry point of the application.

All the functions in *rest.py* have already been described in Chapter 7. We need the *__main__.py* file because it is mandatory for OpenWhisk Python actions. However, for uniformity, in this module there are only some initializations; the actual `main` function is in *control.py*. This is the code for the entry point:

```
import control
import model
def main(args):
    model.init(args["db"], "contact")
    return control.main(args)
```

Before deploying the action, you need to bind a database to a package, as described in "Using the Cloudant Package" on page 210. You also need to create a database and pass the package name of the database to the action. Here, I assume you've already created the package `cruddb` to access the `cruddb` database, and that you've already created the database.

Now we can create the action as a web action and get its URL as follows:

```
$ zip crud.zip __main__.py rest.py model.py view.py control.py
$ wsk action create crud crud.zip \
    --kind python:3 -p db cruddb --web true
ok: created action crud
$ wsk action get crud --url
ok: got action crud
https://openwhisk.eu-gb.bluemix.net/api/v1/web/.../crud/main
```

Of course, your URL will be different because it will have your namespace. Now let's get into the application in detail.

Abstracting Database Access

Let's start by implementing a module that simplifies access to CouchDB or Cloudant. We'll also examine a set of functions for this purpose that we'll put in the *__main__.py* module.

The first problem to solve is the peculiar approach of CouchDB, which uses the `_id` to uniquely identify documents but also requires `_rev` to be able to update or delete a document.

We will try to hide this detail in the module. One way to do this is to use a synthetic id that is a concatenation of `_id` and `_rev`. We assume the user of the module does not need to know of the existence of `_rev` and instead will use only an `id` (without the underscore). We manage encoding and decoding of this `id` internally in the module.

Furthermore, since documents in CouchDB are not typed, it is customary to add a `type` field that helps to select a document in a way similar to classic database tables. All the functions add or search for only documents with a field called `type` that we select in the initialization.

With those assumptions, we are ready to implement our CouchDB abstraction layer, with the following functions:

- `init` for initializing the module
- `insert` for adding new documents
- `update` for updating an existing document
- `delete` for deleting an existing document
- `find` for searching for documents with various criteria

Implementing model.init()

In the `init` function of the database module, we want to select which database we are using and set the type of documents we want to restrict our interactions to. So the first function is `model.init(<database>, <dtype>)`, where *`<database>`* is actually a package we've bound to a specific database and *`<dtype>`* is the document type—in this case just a field called `type` added to each document manipulated by the module.

The implementation of `init` is obvious. We just store the values in two variables in the module that all the functions in the module will use:

```
db = "cruddb"
dtype = "contact"

def init(_db, _dtype):
```

```
global db, dtype
db = _db
dtype = _dtype
```

Implementing model.insert()

The `insert` function receives a JSON object with the fields you want to insert and (optionally) an initial id. If the id is not provided, it is generated. Let's discuss how we handle ids in our abstraction.

In general, you have to remember that the id of a document is not stable: it changes at every operation. So, invoking `insert` on a document returns the new id that you have to use to delete or update it. Similarly, when you `update` a document, it returns a new id that you have to use for further modifications.

Actually, a document is identified by a stable `_id`, but to update it, you also need to use the field `_rev`, which changes at every modification of the document. So we simplify things here: we are using an unstable `id` that is actually a concatenation of the `_id` and `_rev` fields; we return it from `insert` and `update`. We also add a field `id` as the result of searches. In the `insert` function we provide the option to specify the initial id (as it does not require the `_rev` part).

Given those requirements, this is what the `insert` function needs to do:

- Make a copy of the object you are writing to avoid overwriting it.
- Add the current document type (and optionally the id passed as the parameter).
- Invoke the `create-document` action already seen in "CRUD Actions in the Cloudant Package" on page 212.
- Return the new id, concatenating the `_id` and `_rev` values stored in the answer from CouchDB.

This is the code:

```
def insert(args, id=None):
  doc = args.copy()                                          ❶
  doc["type"] = dtype                                        ❷
  if id:                                                     ❸
    doc["_id"] = id
  ret = rest.whisk_invoke( \                                 ❹
    "%s/create-document" % db, \
    {"doc": doc})
  if "ok" in ret:                                            ❺
    return "%s:%s" %\
     (ret["id"], ret["rev"])
  return None                                                ❻
```

❶ Copy the object to avoid overwriting it.

❷ Assign the document type.

❸ Assign the document id, if required.

❹ Create the document.

❺ If the request was successful, create the new id.

❻ Otherwise, return a false Python value.

Before testing the function we need a way to read the data in the database, so first we'll implement the `find` function.

Implementing model.find()

The `find` function can be invoked without arguments. In this case it returns a number of documents (of a given type), up to the pagination limit, which by default is 25. If you instead provide an id, it returns only the document satisfying that id. Note that the id can be either the id returned by a function or the original id (the one returned by `insert` or `update`).

The returned data is also normalized to our way of handling the id, adding explicitly the field concatenating the `_id` and `_rev` values:

```
def find(id=None):
  query = \                                                    ❶
  { "selector": {"type": dtype} }
  if id:                                                       ❷
    query["selector"]["_id"] \
     = id.split(":")[0]                                        ❸
  ret = rest.whisk_invoke(\                                    ❹
    "%s/exec-query-find" % db,\
     {"query": query})
  for rec in ret["docs"]:                                      ❺
    rec["id"] = "%s:%s" %\
      (rec["_id"], rec["_rev"])
  return ret
```

❶ Initialize the query to search all the documents of the current type.

❷ If an id is provided, we add `_id` to the query, and remove the `_rev` part, if any.

❸ This split removes the `_rev` part if it is present.

❹ Invoke the search of the documents.

❺ Normalize the results by adding the `id` field from `_id` and `_rev`.

Testing insert and find

Now we can finally test the functions we wrote using the Python interpreter. As discussed in Chapter 7, we use `doctest` for this, so the interactive interpreter session will be copied into the source code as a documentation comment to become a repeatable test.

The first step in testing is initialization—importing the modules, loading the environment variables to access OpenWhisk, and initializing the `model` module with the database and the record type we want to restrict to:

```
>>> import rest,model,json                                    ❶
>>> rest.load_props()                                         ❷
>>> model.init("demodb","test")                               ❸
```

❶ Import modules.

❷ Load variables to be able to invoke OpenWhisk actions.

❸ Initialize the variables to use a specific database and document type.

Now let's try inserting one record, without specifying any id. As you can see, this generates and returns an id, which we save for future use:

```
>>> args = {"name": "Mike",\
   "email":"msciab@gmail.com"}
>>> id1 = model.insert(args)
>>> id1
'73a395c8020cc6b579f8a7a4c2e2d8a6\
:1-1a96710e8f4c3c9b738eb0250762205f'
```

Now, using `id1`, we can test the `find` function and reload the document we just inserted to see what the database returns. We use `json.dumps()` to render the result in JSON format, to make it easier to read:

```
>>> model.find(id1)
>>> print(json.dumps(
     model.find(id1),\
    indent=2))
{
  "bookmark": \                                               ❶
  "g1AAAABweJzLYWBgYMpgSmHgKy5JLCrJT",
  "docs": [                                                   ❷
    {
      "_id": "73a395c8020cc6b579f8a7a4c2e2d8a6",              ❸
      "_rev": "1-1a96710e8f4c3c9b738eb0250762205f",
      "id": "73a395c8020cc6b579f8a7a4c2e2d8a6:\
      1-1a96710e8f4c3c9b738eb0250762205f",
      "type": "test",                                         ❹
      "email": "msciab@gmail.com",
      "name": "Mike"
```

```
    }
  ]
}
```

❶ The bookmark for pagination.

❷ Documents returned by the query (just one).

❸ The `_id` and `_rev` and the synthetic `id`.

❹ The document `type` and the data just added.

The examples here show actual ids for illustration purposes, but those ids make the test unrepeatable because they change at each operation. The actual test code avoids producing explicit ids by using additional variables. to learn more about how to write tests, I recommend checking out the actual examples in the GitHub repository.

Implementing model.update() and model.delete()

The `update` function works on records returned by `find`. In CouchDB, to update a document you have to provide all the fields to load the document using `find`. Our `find` function also adds an `id`, and the `update` function looks for it.

The implementation works by making a copy of the document (again, to avoid modifying the original), then replacing it with `_id` and `_rev`. The document is then sent to CouchDB for updating, and then we create a new synthetic id. This is the code:

```
def update(args):
  doc = args.copy()
  doc["type"] = dtype                                    ❶
  a = doc["id"].split(":")                               ❷
  del doc["id"]
  doc["_id"] = a[0]
  doc["_rev"] = a[1]
  ret = rest.whisk_invoke( \                             ❸
   "%s/update-document" % db, \
    {"doc": doc})
  if "ok" in ret:                                        ❹
    return "%s:%s" \
     % (ret["id"], ret["rev"])
  return None
```

❶ Ensure the document type is in the answer.

❷ Generate `_id` and `_rev` from `id`.

❸ Actual invocation of the document update.

❹ Generate the synthetic id from the answer.

`delete` also requires an id, but as with `find` we want to use either the synthetic id or the original id. In the first case it is easy—we have both `_id` and `_rev` in the synthetic id. However, if we are using only the original id we do not have `_rev`, so we have to retrieve the document and read the revision before we can delete the document.

Here's how it looks in code:

```
def delete(id):
  a = id.split(":")                                          ❶
  if len(a) == 1:
    res = find(id)                                           ❷
    if res["docs"]:
      a = res["docs"][0]["id"].split(":")                    ❸
    else:
     return {"error": "not found" }
  params = {                                                 ❹
    "docid": a[0],
    "docrev": a[1]
  }
  ret = rest.whisk_invoke(\                                  ❺
   "%s/delete-document" % db, params)
  return ret
```

❶ Check whether the id is a synthetic id.

❷ If not, try to locate the actual id.

❸ Split the synthetic id into components.

❹ Prepare the parameters for deletion.

❺ Delete the document.

Testing update and delete

We can now test the functions in the Python interpreter. Since we are continuing the session from before, we'll reuse `id1`. Now let's test locating the document, then updating and finally deleting it:

```
>>> rec = model.find(id1)["docs"][0]                         ❶
>>> rec["name"]="Michele"                                    ❷
>>> id2 = model.update(rec)                                  ❸
>>> model.find(id2)["docs"][0]["name"]                       ❹
'Michele'
>>> x = model.delete(id2)                                    ❺
```

```
>>> model.find()                                                    ❻
{'bookmark': 'nil', 'docs': [] }
```

❶ Locate the document using the last saved id.

❷ Change a field.

❸ Then update it.

❹ Reload the asset to check the changed value.

❺ Delete the asset.

❻ Check whether it's gone.

This is actually a simplified test. A more detailed test in `doctest` format is available in the repository on GitHub.

The User Interface

The application we are developing has an HTML user interface meant to be used by a web browser. Let's see how the HTML is produced. Given the simplicity of the application, the HTML is generated simply using formatted Python strings. This solution, using multiline strings and the `%` operator, is actually good enough in many cases.

In a more realistic application, you may want to use a templating library to keep the code separate from the HTML markup. This makes it easier to edit using web design tools. In the Python world, the most widely used templating library is probably *Jinja2*.

Recall from Figure 9-1 that there are two main screens in our web user interface: the table and the form. Each one corresponds to a function with the same name. In addition, there is some markup common to both screens, so we have used a `wrap` function to wrap the output of another function. Furthermore, for convenience, since a table is made up of rows, we define a `rows` function that only renders the rows of the table.

Testing

We also need to figure out how to test the HTML output. In this section, you'll see how to test the output with the help of the library Beautiful Soup.

While the user interface functions produce a lot of markup, it is in large part fixed, and we're interested in testing only those parts that are calculated. Most notably, we want to test whether those parts of the HTML containing variables actually change as expected. `BeautifulSoup`, to ease testing, helps in selecting only those snippets of the HTML markup you want to verify.

The wrapper

Let's look at the `wrap` function. The function itself is not very interesting—it just produces an HTML string—but it's useful to illustrate testing techniques:

```
BOOTSTRAP_CSS = "https:/.../bootstrap.min.css"
BOOTSTRAP_JS = "https://.../bootstrap.min.js"
JQUERY = "https://.../jquery.min.js"

def wrap(body):
    return """<!DOCTYPE html>
<html lang="en">
  <head>
    <title>OpenWhisk Crud Demo</title>
    <link rel="stylesheet" href="%s">
    <script src="%s"></script>
    <script src="%s"></script>
  </head>
  <body>
   <div class="container">%s</div>
  </body>
</html>
""" % (BOOTSTRAP_CSS, \
 JQUERY, BOOTSTRAP_JS, body)
```

As expected, there's nothing special here. But let's see if the markup produced actually replaces the parameters correctly. Of course, we can print the entire HTML and compare the result, but this usually leads to tests that are difficult to read and update.

BeautifulSoup

A better approach is to focus just on the important parts, and here is where `BeautifulSoup` comes in. This Python library is designed to manage HTML (and XML) markup. It works by parsing HTML markup using one of the many available Python parsers (the recommended one is `lxml`). Once parsed, the HTML is transformed into a hierarchy of Python objects that you can then inspect. You also get a set of methods like `find` and `find_all` to locate subtrees in the parsed markup.

Let's use `BeautifulSoup` to test the `wrap` function. As usual, we use an interpreter session to prepare our tests, then copy and paste the session into a docstring so we can repeat them later with `doctest`:

```
>>> import view                                                    ❶
>>> from bs4 import BeautifulSoup as BS                           ❷
>>> html = BS(view.wrap("BODY"), "lxml")                          ❸
>>> print(html.body.div)                                          ❹
<div class="container">BODY</div>
>>> print(html.find_all("link")[0])                               ❺
<link href="https://.../bootstrap.min.css" rel="stylesheet"/>
```

❶ Import the module we want to test.

❷ Import the `BeautifulSoup` parser.

❸ Invoke the `wrap` function and parse the result.

❹ Extract the main `DIV` of the wrapper to verify the replacement.

❺ Search the document to locate a tag with replacements.

Note how we used the `BS` function to parse our markup. The `"lxml"` parameter is the parser to use (but there are other parsers that can be useful in specific cases). The result is a Python object with fields and subfields corresponding to the HTML structure of the parsed markup. The hierarchy of objects corresponds to the hierarchy of tags: under `html` the main tag is `body`, which in turn it has the subtag `div`; we print it to be sure there is a replacement inside.

The `html.find_all("link")` returns an array of all the possible tags of this type. Since we expect only one, we printed only the first result to verify that the replacement happened correctly.

Rendering the Table with view.table

The main view of our CRUD application is a table with the records we found in the database. We'll first write the method `rows`, which expects an array of maps coming from the database. When we query the database we get something like this:

```
[
  {
   "id": "xxx:yyyy"
   "name": "Mike",
   "email":"msciab@gmail.com"
   },
   {
   "id": "zzz:ttt"
   "name": "Miri",
   "email":"miri@gmail.com"
   }
]
```

This answer translates to an array of Python maps. We are going to render it in HTML using a cell for each value and a radio button for the id, which is used for editing and deleting a document. The `rows` function hence is:

```
def rows(docs):
  res = "<tbody>"
  for row in docs:
    res += """
 <tr>
  <td scope="row">
   <input name="id" value="%s" type="radio">
  </td>
  <td>%s</td>
  <td>%s</td>
 </tr>
""" % (row["id"], row["name"], row["email"])
  return res+"</tbody>"
```

The function is straightforward, so we can test it using the same techniques we used before. Note that we leverage the fact that the id handling is hidden in the `model` functions, so we use it without changes.

Now let's see the `table` function that wraps `rows`. This function generates a form that encloses all the radio buttons. Furthermore, there are buttons for things like adding a new record (`op=new`), editing the selected one (`op=edit`), or deleting a record (`op=delete`). The code is as follows:

```
def table(data):
    res = ""
    res += """
<form method="post">                                                    ❶
 <table class="table">
  <thead>
    <tr>
     <th scope="col">#</th>
     <th scope="col">Name</th>
     <th scope="col">Email</th>
    </tr>
  </thead>"""
    res += rows(data)                                                   ❷
    res += """
  <tfoot>
    <tr>
     <td colspan="4" align="center">
       <button name="op" value="new"                                    ❸
        type="submit" class="btn btn-default">
         New Contact</button>
       <button name="op" value="edit"                                   ❹
        type="submit" class="btn btn-default">
         Edit Contact</button>
       <button name="op" value="delete"                                 ❺
```

```
        type="submit" class="btn btn-default">
         Delete Contact</button>
      </td>
    <tr>
   </tfoot>
  </table>
</form>"""
    return res
```

❶ A form enclosing all the records.

❷ Render the actual data of the table.

❸ Create a new record.

❹ Edit the selected record.

❺ Delete the selected record.

Rendering the Form with view.form

The `form` function is a bit more interesting. It is invoked in two different cases: when you want to create a new record and when you want to update one. The main difference is that for new records we do not specify the id since the `insert` function can generate one. When editing a record instead we add a hidden input field, storing the id of the current record needed to perform the update:

```
def form(args):
  id = ""
  if "id" in args:
    id = """
     <input type="hidden"
      name="id" value="%s">
     """ % (args["id"])
  return """
<form method="get">
  %s
  <input type="hidden"
   name="op" value="save">
  <div>
    <label for="usr">Name:</label>
    <input id="name" name="name"
     type="text" value="%s">
  </div>
  <div>
    <label for="email">Email address:</label>
    <input id="email" name="email"
     type="email" value="%s">
  </div>
  <button type="submit">Save</button>
```

```
</form>
    """ % (id, args["name"], args["email"])
```

The code is simplified for readability (I removed formatting attributes). I also don't include tests here because they are pretty obvious and straightforward.

The Controller

The controller is the key component of this simple application, and probably the most interesting. It puts together the model logic with the view layer we discussed. It is OpenWhisk-specific, since it works with the action-based serverless model, reading inputs from `args`. It is also the most difficult part to test because we need to use mocking to verify the code locally.

The controller consists of the function `main` in the module *control.py*, and the support function `fill` that initializes a Python dictionary from the arguments to store it in the database. Let's look at the `fill` function first. It exists solely to create an object with the relevant fields for the database. Indeed, in the `args` objects there are many other values that you may not want to store in the database:

```
def fill(args):
  res = { }
  if "id" in args: res["id"]=args["id"]                    ❶
  res["name"] = args.get("name", "")                       ❷
  res["email"] = args.get("email", "")
  return res
```

❶ Copy the id only if it is available.

❷ Initialize other fields to the passed value or a default.

Next is the function `main`, which begins with some initializations and processes all the operations. If no operations are specified (as happens when you invoke the actions without parameters) it just renders the main table:

```
def main(args):
    op = args.get("op")                                    ❶
    # ...processing operations...                          ❷
    data = model.find()["docs"]                            ❸
    body = view.table(data)                                ❹
    return { "body": view.wrap(body) }                     ❺
```

❶ Get the current `op` from the field defined above (`<input type="hidden" name="op" value="save">`) to start processing it.

❷ Process operations as described later.

❸ You get here if there are no other operations, so the default is to read all the documents.

❹ Fill the table with the data.

❺ Render the web action, wrapping the table.

To do something different than just showing the records, the controller needs to respond to form submissions. Each form submission sets the variable `op` and other fields. When you click a button, it is always named `op` and its value is used to decide what to do. The possible values for `op` hence are:

- `new`
- `edit`
- `delete`
- `save`

In the next sections, we'll develop our application further to handle each of these cases.

Processing Operations

First, let's process the operations. If the current operation is `new`, you just need to invoke the form with an empty value:

```
if  op == "new":                                          ❶
    form = view.form(fill({}))                            ❷
    return { "body": view.wrap(form) }                    ❸
```

❶ Capture the operation `new`.

❷ Fill the form with an empty body.

❸ Return the wrapped answer as a web action body.

The `edit` operation is a bit more complicated.

First, the operation is possible only if you selected a record to edit. This means you clicked on a radio button to select a record. If you did so, the current value of `id` lets you retrieve the record from the database. You load it and use the value to fill the form. Note that when you use `find` to specify an `id`, you only have to retrieve the first document from the answer:

```
if op == "edit" and "id" in args:                    ❶
    res = model.find(args["id"])                     ❷
    rec = res["docs"][0]                             ❸
    body = view.form(rec)                            ❹
    return { "body": view.wrap(body) }               ❺
```

❶ Capture a valid `edit` operation.

❷ Load the data from the database.

❸ Extract the record (there should be exactly one).

❹ Use the value to fill the form.

❺ Return the wrapped answer.

So, with the `new` operation you see a form that is empty (with the `id` not set), while with the `edit` operation you see a form that is prefilled.

On the form screen, you can edit the values and then click the Save button. What happens now depends on whether the record is new or you are updating an existing record. You can distinguish between the two cases because there is no `id` when the record is new. If there is an `id`, you want to update the record; otherwise, you want to insert it.

Let's see the code to process a `save` operation:

```
if op == "save":                                     ❶
  if "id" in args:                                   ❷
    model.update(fill(args))                         ❸
  else:
    model.insert(fill(args))                         ❹
```

❶ Determine if you are saving.

❷ Is this a new record or an existing one?

❸ Existing: fill in the data from the `args` and invoke the `model.update` function.

❹ New: fill in the data from the `args` and invoke the `model.insert` function.

Now let's see how to delete. First, the operation is possible only if you've selected a record to edit. This means you clicked on a radio button to select a record. The implementation is just a matter of invoking `model.delete`:

```
if op == "delete" and "id" in args:                  ❶
   model.delete(args["id"])                          ❷
```

❶ We want to delete *and* we have an id.

❷ Just do it.

Testing the controller using mocking

Now let's test the controller using the usual the `doctest` approach and the Python interpreter. The main difference here is that we are going to use mocking. The controller uses the model, which in turn talks with the database. In general, however, we are not interested in verifying that the model functions work (as they were already tested), but just that the controller reacts properly to the various requests. This is a use case for mocking: in your tests, you just want to see what happens when the function is invoked, providing some fixed results from the database functions, which are already encapsulated in a module.

Using Python, a highly dynamic language, you can perform mocking without using external libraries (at least in simple cases). There is a complex and powerful mocking tool in the standard library, but here we just need to replace some functions in an imported module with mocked functions returning fixed values.

Let's start by importing the model and `BeautifulSoup` to check the HTML output:

```
>>> import model
>>> import control
>>> from bs4 import BeautifulSoup as BS
```

Now let's see what happens when we invoke the controller the first time, without any defined operations. If you look at the code, you'll see that in this case it queries the database in order to retrieve all the records of our current type: `data = model.find()`.

But we don't actually want to invoke the database! Instead, let's mock the `model.find()` function. In Python, tis is extremely easy: we just assign it to a function returning the value we expect. To verify the result, we just want to look into the output and see if the body of the table contains our mocked data:

```
>>> docs = [{"id":"1","name":"Mike","email":"m@s.c"}]                    ❶
>>> model.find = lambda x=None: {"docs":docs}                            ❷
>>> res = control.main({})                                               ❸
>>> html = BS(res["body"], "lxml")
>>> print(html.find("tbody"))                                            ❹
<tbody>
<tr>
<td scope="row">
<input name="id" type="radio" value="1"/>
</td>
<td>Mike</td>
<td>m@s.c</td>
```

```
</tr>
</tbody>
```

❶ Mocked data.

❷ Mock `find` with a function returning fixed data.

❸ Invoke the controller without arguments.

❹ Check that the returned data matches the mocked data.

As expected, the output contains one row with the data returned from the mock. A more realistic test would include the empty case and multiple rows, but that is omitted here for brevity. The next step is to test `new` operations.

There are no database operations involved—just make sure the output produces a form with empty fields:

```
>>> res = control.main({"op":"new"})
>>> inp = BS(res["body"], "lxml").find_all("input")
>>> print(*inp, sep="\n")
<input name="op" type="hidden" value="save"/>
<input class="form-control" id="name" name="name" type="text" value=""/>
<input class="form-control" id="email" name="email" type="email" value=""/>
```

Now, let's test the `edit` operation. The main difference here is that it is going to use data from the database using `find`. Since `find` is already mocked, we can expect that executing an `edit` operation will return the input fields initialized with the values in the array of docs we provided:

```
>>> res = control.main({"op":"edit", "id":"1"})
>>> inp = BS(res["body"], "lxml").find_all("input")
>>> print(*inp, sep="\n")
<input name="id" type="hidden" value="1"/>
<input name="op" type="hidden" value="save"/>
<input class="form-control" id="name" name="name" type="text" value="Mike"/>
<input class="form-control" id="email" name="email" type="email" value="m@s.c"/>
```

Side Effects

Next, we will test the `save` operation. Here again, we need to use mocking—this time with the `insert` and `update` operations.

In this case, we want to check not just the value returned, but what happened as a side effect. In particular, we want to know which value will be written in the database. Again, in Python this is extremely easy: just define a function that stores the result of an invocation in a variable, then read that variable.

Let's write the `inspect` function, saving the received value as `spy` in the `control` package:

```
spy = {}                                    ❶
def inspect(x):                             ❷
  global spy                                ❸
  spy = x.copy()                            ❹
```

❶ A variable to store the inspected value.

❷ The function to use to capture the values.

❸ Access the global variable.

❹ Copy our value in the global variable.

Now, all we need to do is replace the functions we want to inspect with our `inspect` function, then see what happened by looking at the `spy` variable. Let's first test the `insert` operation:

```
>>> model.insert = control.inspect          ❶
>>> args = {"op":"save", \                  ❷
     "name":"Miri", "email":"m@d.g"}
>>> x = control.main(args)                  ❸
>>> print(control.spy)
{'name': 'Miri', 'email': 'm@d.g'}          ❹
```

❶ Replace `insert` with our inspector.

❷ Prepare a `save` operation without ids.

❸ Invoke the `save` operation with our values.

❹ Print the `spy` variable in order to verify that the values are as expected.

We can do the same for `update`, but here we have to set an id:

```
>>> model.update = control.inspect          ❶
>>> args = {"op":"save", "id": "1",\        ❷
     "name":"Mike","email":"m@s.c"}
>>> x = control.main(args)                  ❸
>>> print(control.spy)
{'id': '1', 'name': 'Mike', 'email': 'm@s.c'}
```

❶ Replace `update` with the inspector.

❷ Trigger an update using the id.

❸ Check the result.

Advanced Web Actions

Recall from Chapter 2 that you can create a web action with the `wsk create` tool using the flag `--web true` and then read its URL (to be used in a browser) with the `wsk get` command and the `--url` flag.

In general, a web action receives a request in the form of a JSON object with some standard fields providing details of the HTTP requests, and it must produce a response in the form of another JSON object; this JSON object must contain some required fields that have a special meaning in producing the HTTP response.

First, let's discuss the response fields; later in this section we'll work on the request fields. The response's required fields are:

`body`
: The body of an answer, a string; it is usually HTML but it can be plain text or a base64-encoded string when the body is binary.

`statusCode`
: The HTTP status code, also a string; `200` means OK, while other values, as mandated by HTTP standards, mean some special action (like a redirection) or some kind of error.

`headers`
: A map, where the keys are header names and the values are header values.

Now let's use this knowledge to write an example, *echoweb.py*, which returns the request as a JSON object so that we inspect it.

```
def main(args):
  return {
    "body": args,                                          ❶
    "status": "200",                                       ❷
    "headers": {
      "Content-Type": "application/json"                   ❸
    }
  }
```

❶ The body contains the arguments.

❷ The status is `200`, the HTTP OK code.

❸ Return the content type declaring the answer is a JSON object.

Now you can deploy it and get a URL:

```
$ wsk action update python/echoweb echoweb.py \
  --web true --kind python:3
ok: updated action python/echoweb
```

```
$ URL=$(wsk action get python/echoweb --url| tail -1)                    ❶
$ echo $URL
https://openwhisk.eu-gb.bluemix.net/api/v1/web/.../python/echoweb
```

❶ Store the URL of the `echoweb` action in the `$URL` variable.

Now you have an action you can use to explore the format of requests. Before getting into a practical example, let's look at the fields in the request and their meanings:

- `__ow_method` is the HTTP method of the request (`GET`, `POST`, `PUT`, etc.).
- `__ow_path` is the "extra" path of the request.
- `__ow_headers` is a map of the headers of the request.
- `__ow_body` is the body of the request.
- `__ow_user` is the user, available only if there is an annotation password-protecting the request.

Let's try a simple request using `curl` and `$URL` as is (i.e., using `GET`). In this case there is no body; there is just an empty extra path and there are no additional values. (note that the output has been simplified to remove unnecessary details):

```
$ curl $URL
{
  "__ow_method": "get",                                                  ❶
  "__ow_path": "",                                                       ❷
  "__ow_headers": {                                                      ❸
    "user-agent": "curl/7.54.0",
    "host": "...",
    "accept": "*/*",
    "accept-encoding": "gzip",
    "x-forwarded-port": "443"
  }
}
```

❶ The method of the request is `GET`.

❷ There is no path after the action name.

❸ Additional information on the request available as headers.

Now let's try a more complicated request: we are going to create a request using `POST` that sends a form with two fields in URL-encoded format and specifies an extra path and a query string.

This is the result, also simplified for readability:

```
$ curl -X POST -d 'user=mike&pass=hello' $URL/hello?q=world
{
  "__ow_method": "post",                                                 ❶
```

```
  "__ow_path": "/hello",                                                    ❷
  "pass": "hello",                                                          ❸
  "user": "mike",
  "__ow_headers": {
    "content-type": "application/x-www-form-urlencoded",                    ❹
    "host": "...",                                                          ❺
    "accept": "*/*",
    "user-agent": "curl/7.54.0",
    "accept-encoding": "gzip",
    "x-forwarded-port": "443",
    "x-client-ip": "...",
    "x-forwarded-proto": "https",
    "x-real-ip": "...",
    "x-forwarded-host": "...",
    "x-forwarded-for": "..."
  },
  "q": "world"                                                              ❻
}
```

❶ The method this time is `POST`.

❷ The "extra path" after the URL.

❸ Parameters of the form, already decoded.

❹ The content type, a URL-encoded form.

❺ Miscellaneous information available in the headers.

❻ This parameter comes from the query string.

Improving the CRUD Application

Now let's improve the application by adding some more advanced features. We'll also discuss web actions in more depth here, because we'll use them to implement some of these advanced features. The features we are going to add include:

- Data validation
- Pagination of output
- File upload
- Image rendering

Data validation is performed on the server side using database data validation features. Pagination also relies on database features, in particular bookmarking. File upload and image rendering instead require advanced web actions.

Figure 9-3 shows the application as it will appear at the end of the chapter.

For reference, all the changes are available on GitHub.

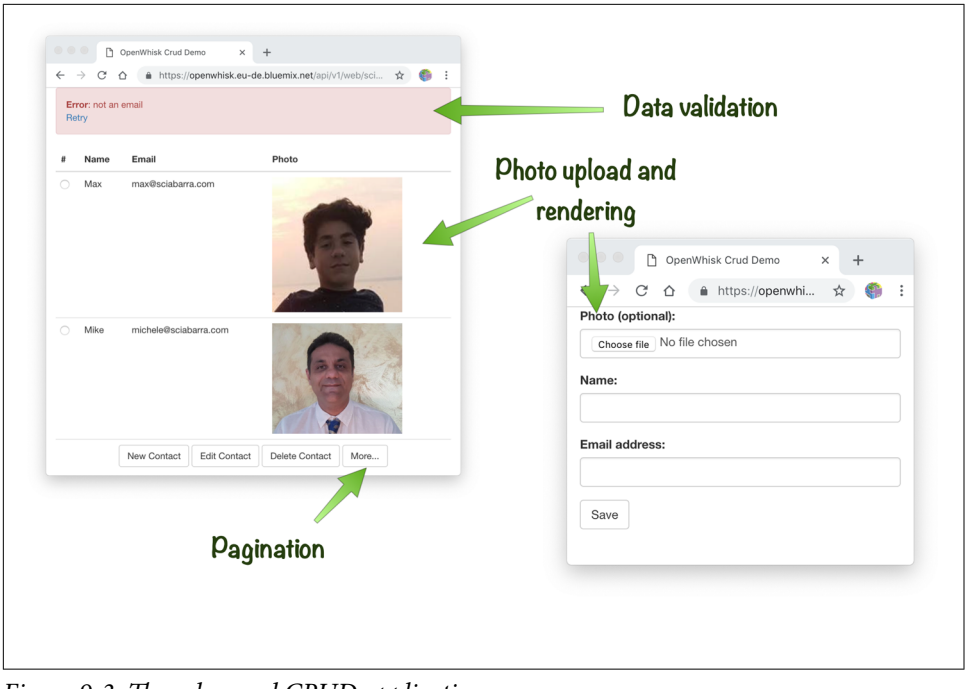


Figure 9-3. The advanced CRUD application

Validation and Error Reporting

At this point, our CRUD application is missing validation of the input. Let's now add server-side validation of the data, with the help of the database. In "Validation Functions" on page 208 we discussed the design document *validate.json*, which is also suitable here. The validation function is a JavaScript function, in *validate.js*, that checks if in a document with `type='contact'` that both the name and email address are defined. We are going to reuse this function, but first let's see how to deploy this design document using the `wsk` tool (in the other example we used `curl`). For deployment we also need to prepare a JSON object as a `wsk` parameter file using `jq`:

```
$ jq -n --rawfile file validate.js \
  '{ "doc":{ "_id": "_design/validate",\
     "validate_doc_update":$$file} }' \
```

❶

```
  >validate.json
$  wsk action invoke \
   advcruddb/create-document \                    ❷
   -P  validate.json -r                           ❸
{
    "id": "_design/validate",
    "ok": true,
    "rev": "1-b2803e26e4d30c3e458908df02192150"
}
```

❶ Prepare the JSON with the parameter `doc`.

❷ Invoke the `create-document` action.

❸ Here the `doc` parameter comes from the JSON.

Now we have our validation function in place. If you try to run the current application without changes, you will not be able to save empty records or wrong emails, but the application does not show any error messages. This is the feature we need to work on now—the ability to throw errors. We need to make the following changes to do this:

- In *model.py*, store the latest error in a well-known place (a module variable).
- In *view.py*, read a parameter to display errors.
- In *control.py*, propagate error messages to the view.

Storing Error Messages

In *model.py*, we add a variable containing the latest error:

```
# add this after the imports
last_error = None
```

Then, we store the error message when something goes wrong. Note that since the variable `last_error` keeps track of the last error, if there are no errors, we reset it.

```
 # add this after
 # ret = rest.whisk_invoke(...
 # in insert and update
  global last_error
  if "error" in ret:
    last_error = ret["error"]["message"]          ❶
  if "ok" in ret:
    last_error = None                             ❷
    return "%s:%s" % (ret["id"], ret["rev"])
```

❶ Save the error if there is one.

❷ Clean the error if there is none.

The error message is now saved in the `module.last_error` variable, but to render it you need to change *view.py*; in particular, you have to edit the function `table`. You go to the table screen after any operation that goes wrong, so it is the right place to show the error. For this purpose, we add a parameter `error` to the `table` function:

```
# modify the table function
def table(data, error=None):
    res = ""
    if error:
      res += """
<div class="alert alert-danger"
 role="alert"><b>Error</b>: %s <br>                    ❶
 <a href="javascript:window.history.back()">Retry</a>  ❷
</div>""" % error
    # rest of the form
```

❶ Display the error.

❷ Go back to the form to fix it if there is an error.

The last step is to change *control.py* to propagate errors from the model to the view:

```
# modify the end of the main function
# to pass model.last_error to view.table
body = view.table(data, model.last_error)   ❶
if model.last_error:
    model.last_error = None                 ❷
return { "body": view.wrap(body) }
```

❶ Pass the error to the view.

❷ Clean the error after use to avoid multiple error messages.

Pagination

We've come a long way, but our application still has one issue: when it displays the records, it displays only the first 25. This is because by default any search has a default limit of 25. We could increase the limit to show more data, but it would be better to implement paginated visualization. We will do this using the bookmark feature discussed in "Pagination Support" on page 194.

For simplicity, we'll show only forward pagination: we'll add a More button that displays the next page until we reach the last page of data. Then, the button changes to Restart to display the first page again.

The implementation first requires a few changes to the model:

- Set the number of rows you fetch when you search.
- Add an index to the database to retrieve ordered data.
- Use the bookmark in queries to get the next page.

Once you can get paginated data, the other changes are pretty simple:

- The view must be modified to display the pagination buttons.
- The control just needs to pass the bookmark around.

Creating an Index

To get consistent and efficient pagination, you need to index your data, as described in "Indexes" on page 192. We'll create an index that will maintain an ordered list of names. Indexes are created using design documents, so here we create a design document to index our contacts by `name`:

```
{
  "index": {
    "fields": [
      {
        "name": "asc"
      }
    ]
  }
}
```

To publish this index, we have to invoke the `create-query-index` action. We pass the whole index document as the parameter `index`, so we again create a parameter file using `jq` and then invoke the action with `-P`:

```
$ jq -n --slurpfile index index.json \                          ❶
  '{"index": $$index[0]}' >_index.json                          ❷
$ wsk action invoke \
   advcruddb/create-query-index \                               ❸
   -P _index.json -r
{
    "id": "_design/c48f920d130eea44840366ac64cde81858bed902",
    "name": "c48f920d130eea44840366ac64cde81858bed902",
    "result": "created"
}
```

❶ Read the index file.

❷ Write the parameter file.

❸ Create the index with the parameter file.

Using Bookmarks and Limits

The `find` function is all about running a query against the database. In the first example the query was basic: we did not specify any ordering, we used the default limit, and we only filtered by document type.

Now, we want to do better. We will:

- Set the limit explicitly.
- Specify some ordering of the results.
- Get a specific page using a bookmark.

The most important changes are to the search query. There is a module variable, initialized to `10`, that defines the number of rows shown on each page. We use this to limit the number of rows returned. We then change the query to order the results by name and set the limit. But most importantly, we use the bookmark so we can start the search from the current page:

```
find_limit = 10                                                    ❶
def find(id=None, bookmark=None):                                  ❷
  # ...
  global find_limit
  query = {
    "selector": {"type": dtype},
    "sort": [{"name": "asc"}],                                     ❸
    "limit": find_limit                                            ❹
  }
  if bookmark:
    query["bookmark"] = bookmark                                   ❺
  # ...
```

❶ How many documents the `find` must retrieve.

❷ Add a new parameter, `bookmark`, that defaults to `None`.

❸ Ask to sort results by `name`.

❹ Limit the number of returned documents.

❺ If the `bookmark` is defined, add it to the query.

Pagination

Since the data is actually returned paginated, there are no changes in the visualization of rows. We just need a button to move to the next page. The view needs to know the value of the current bookmark, too, so we have to pass it. This value is available in the

dictionary returned by `find`. We use it to set the value of the More button to advance to the next page. If the page is the last one, we instead display a Restart button and use an empty bookmark:

```
def table(data, bookmark=None, error=None):                    ❶
  # ...
  if bookmark:                                                 ❷
    button = """
    <button name="bookmark" value="%s"                         ❸
    type="submit">More</button>
      """ % bookmark
  else:
    button = """
    <button name="bookmark" value=""                           ❹
    type="submit">Restart</button>"""
  # ...
  print "...%s..." % button
```

❶ Pass a bookmark as a parameter when displaying the table.

❷ If the bookmark is defined, put it as the value of a button.

❸ Set a button with `name=bookmark`.

❹ Otherwise add a button with an empty bookmark.

Processing the Bookmark

Normally when you paginate data you have to keep track of current indexes; using bookmarks, all of this is done for you by the database. The bookmark embeds all the required information and all you need to do is search, passing the `bookmark` parameter.

We only have to take care of one thing: understanding when we are at the last page. As a simple criterion (it can be more complicated), we can detect that we are at the end of the pagination when the search returns less documents than the limit.

So, we need to:

- Replace data loading in the basic *model.py* by reading an additional parameter (`bookmark`).
- Search in the database using it.
- Determine whether we are at the last page.

  The following code does this:

```
# ... replace this after 'data = ...'
# paginated rendering
```

```
curr = args.get("bookmark")                                    ❶
query = model.find(bookmark=curr)                              ❷
data = query["docs"]
next = ""                                                      ❸
if len(data) == model.find_limit:                              ❹
   next = query.get("bookmark")
body = view.table(data, next, model.last_error)                ❺
```

❶ Get the current bookmark as a parameter.

❷ Search using it (it can be `None` or empty).

❸ Prepare for the next page.

❹ If we are not yet at the last page, read the bookmark.

❺ Render the current page, but pass the bookmark of the next one.

Uploading and Displaying Images

Next, let's add the ability to upload a photo and display it. This is probably the most complicated part of this small application, so I'll explain it in small steps.

In this demo application we store the image in a field of the database as a base64-encoded string, and then we retrieve the image from the database using an action. I must to make 100% clear that this is *not* a good idea (except for very small amounts of data). You should use instead some form of object storage like Amazon S3, or similar services available in every cloud. This is meant to be a simple example. We already have the methods to store and retrieve data from our database, so using it for the images, too, was very easy.

Let's consider what it takes to upload images:

- The data entry form needs a new field for image upload.
- The controller will receive data from a file upload that it must parse to prepare the data for storage.
- The table must include an `<IMG>` tag to show photos.
- The controller must handle requests to render photos.

Luckily, *model.py* does not need changes. We just have a document with more fields; the existing code is fine for reading and writing in the database. Most of the complexity is actually hidden in the `form_parse` method that handles the details of form pars-

ing. Furthermore, since in the standard Python library there is not a parsing function for file uploads (except a deprecated one), we use a third-party open source library (included in the source code of the action). Other details on preparing data to be stored in the database are handled by the `fill` function.

Let's discuss all those things in order.

File Upload Form

First, let's change the function `form` in *view.py*. In the form, you need to make two changes. The first is the obvious addition of the tag `<input>` of type `file` to upload images. The second change is the encoding of the form. When you use the form to upload, the HTML rules require using the method `POST` and setting the encoding to `multipart/form-data`. So let's change the HTML returned by the view:

```
    # ...
    return """
<form method="post"
  enctype="multipart/form-data">                                    ❶
  <!-- ... -->
  <label for="photo">Photo (optional):</label>
  <input type="file" class="form-control"
    id="photo" name="photo">                                        ❷
   <!-- ... -->
</form>""" # ...
```

❶ Encode the form to upload a file.

❷ Add a tag to upload a file.

Now that the form can upload, a `submit` sends to the controller a body encoded as `multipart/form-data`. Since we can't rely on the parameters anymore, leveraging the knowledge gathered in "Advanced Web Actions" on page 239, we parse the `__ow_body` parameter that is now set to the *encoded* form upload.

In the controller, we are no longer using the condition `op == "save"` to save data. Instead, we check to see if the `__ow_body` has the uploaded data to parse and then we invoke a new `form_parse` function (examined later) to read the data.

This function returns two dictionary-like objects—one for normal fields and another one for file uploads. You have to extract data in a special way for files (we handle those details later, when we update the `fill` method):

```
# add this at the beginning of `main`
if "__ow_body" in args:                                             ❶
  fields, files = form_parse(args)                                  ❷
  filled = fill(fields, files)                                      ❸
  if "id" in fields:                                                ❹
    model.update(filled)
```

```
    else:
      model.insert(filled)
```

❶ Since `__ow_body` is available, we have to parse it.

❷ Parse the form, getting fields and files.

❸ Fill the form with the parsed data.

❹ Do an insert or an update as usual.

Parsing the File Upload

In the standard Python library there is only a deprecated parser for the data of a file upload. So here, we use a third-party library to parse it: `multipart`. The library is included in this code for the chapter on GitHub. I've also forked the original repository for reference.

This library requires some information before processing. Luckily, all the informations that it needs is available in a web action. Parsing a file upload requires the following:

- The HTTP method, available in `__ow_method`
- The content type, available in the map `__ow_headers` under `content-type`
- The body of the request, available in `__ow_body`

A few more things to note:

- The `content-type` is critical for parsing, as it contains an unique string (the *boundary*), essential for parsing the body.
- The `__ow_body` is actually encoded in base64 so must be decoded first.
- The function `multipart.parse_form_data` expects the input using a "file-like" object, so we create one with `io.BytesIO`.
- Everything is encoded in `UTF-8`.

Here's the definition of the `form_parse` function:

```
def form_parse(args):
    import io, base64, multipart
    body = args.get("__ow_body")                                    ❶
    input = io.BytesIO(base64.b64decode(body))
    method = args["__ow_method"]                                    ❷
    ctype = args["__ow_headers"]["content-type"]                    ❸
    env = {                                                         ❹
        'REQUEST_METHOD': method,
```

```
        'CONTENT_TYPE': ctype,
        'wsgi.input': input
    }
    return multipart.parse_form_data(env, \                    ❺
            strict=True, charset='utf-8')
```

❶ Read the body, decoding from base64, then encoding as an in-memory file.

❷ Read the method.

❸ Read the content type.

❹ Prepare a map to pass those parameters to the parsing function.

❺ Actual parsing of the `multipart/form-data` upload.

The function returns two dictionary-like objects: one containing the normal fields, and another, more complex object containing file uploads (that you have to read) and additional information like the content type. Those objects are used in the next section to save data in the database.

Saving Data in the Database

Now we have to prepare a document, using the parsed form, to write it in the database. A web action returns binary data (like images) in base64 format and also must return the content type. We are going to store this information in just two fields: one named `photo`, with data encoded in base64, and another named `photo_mime` containing the MIME type of the upload. The result is an ordinary document that can be saved using the available model functions.

The only difficulties are the extraction of the data from the dictionary object. In particular, we have to read a file-like object. Then we encode it as a base64 array of bytes, and finally decode it to an ASCII string for saving:

```
def fill(fields, files):
  res = { }
  if "id" in fields: res["id"]=fields["id"]                    ❶
  res["name"] = fields.get("name", "")
  res["email"] = fields.get("email", "")
  if "photo" in files:                                         ❷
      from base64 import b64encode
      photo = files["photo"].file.read()                       ❸
      res["photo"] = b64encode(photo).decode('ascii')          ❹
      res["photo_mime"] = files["photo"].content_type          ❺
  return res
```

❶ Read the id, name, and email address as usual.

❷ Check if there is a photo uploaded.

❸ Read the photo as a file object, returning bytes.

❹ Encode the photo in base64 and extract it as an ASCII string.

❺ Read also the MIME type that was provided in the upload.

Generating an <IMG> Tag

Since in the database the contacts are documents, and they now include images encoded in base64, an image is stored as an ordinary field (not as an attachment). This is easy and convenient both for saving and loading data. However, to display the image with a web browser you have to do things differently.

In HTML you display images using the tag `<IMG SRC="<url>">`, where *<url>* is a URL that returns an image. When using an ordinary web server, to place a static image somewhere, you use a URL pointing to the image URL within the web server. In OpenWhisk there is not a place to save static images, so you have to serve the image by yourself using an action.

Here, we are going to serve images using the same action. To do so, we use the extra path, a feature of web actions discussed in "Advanced Web Actions" on page 239.

When you invoke a web action it has a URL in this format:

```
https://<hostname>/api/v1/web/<action-name><extra-path>
```

The *<action-name>* is enough to locate the action, but you can also add the *<extra-path>* part; it is entirely optional and is passed as a parameter to the action in the `__ow_path` variable.

This handling of images works by generating URLs for images, then concatenating an extra path in the format */<id>* to the complete URL of the action. In the controller, then, we must check the variable `__ow_path`, where we can read this part. If the *<extra-path>* is empty we serve the `main` action, producing the HTML you have seen so far. If it is not empty, we have to retrieve an image. In this case, we have to return the image as the body, specifying the content type and the image encoded in base64.

But let's do this in order: first we'll see how to generate the `<IMG>` tag in *view.py*, then we'll change *controller.py* to render an image as extracted from the database.

Generating a URL to Retrieve an Image

We have to modify the `rows` function in the *view.py* module to retrieve the images. At present, this function only displays the name and email address. Now we want it to

also display uploaded images. We need to generate an `<IMG>` tag with an appropriate URL that also includes the `_id` of the document. The next listing shows the entire updated `rows` function. It is a bit long, but hopefully simple to understand.

Here are a few things to keep in mind:

- We don't specify the hostname, so the URL is relative to the current hostname.
- It starts with `/api/v1/web` so the URL uses an absolute path within the current host.
- It reads the current action name from the environment variable `__OW_ACTION_NAME`.
- It specifies the `_id`, removing the `_rev` part from the `id`.

Here's the new `rows` function:

```
def rows(docs):
  res = "<tbody>"
  for row in docs:
    img = ""
    if "photo" in row:
      action = os.environ["__OW_ACTION_NAME"]                    ❶
      _id = row["id"].split(":")[0]                              ❷
      url = "/api/v1/web%s/%s" % (action, _id)                   ❸
      img = '<img width="200" src="%s">' % url                   ❹
    res += """
 <tr>
  <td scope="row">
   <input name="id" value="%s" type="radio">
  </td>
  <td>%s</td>
  <td>%s</td>
  <td>%s</td>                                                    ❺
 </tr>
""" % (row["id"], row["name"], row["email"], img)                ❻
  return res+"</tbody>"
```

❶ Get the action name.

❷ Get the `_id`.

❸ Construct the URL for the image.

❹ Produce the `<IMG>` tag with the URL.

❺ A new cell in the table for the image.

❻ Also pass the tag URL to insert it in the table.

A simpler and more efficient way of rendering images involves using a `data:` URL. Using an `<IMG>` tag in the format: `<IMG SRC="data:<content-type>;base64,<image-body>">`, you can embed the images in the HTML and render the image without an additional HTTP request. We follow the more complex path of serving images with *`<extra-path>`* in order to demonstrate more features of web actions.

Rendering the Image with an HTTP Request

The HTML of the action now includes an `<IMG>` tag like the following:

```
<img width="200"
  src="/api/v1/web/...\
  /advcrud/main/4ac118ee9b8389439cb3cf0954b07c7c">
```

When the browser tries to render this tag, it invokes the action again for each image, setting the value `/4ac118ee9b8389439cb3cf0954b07c7c` in the `__ow_path` variable. So we have to handle it, returning the photo embedded in each document.

Our code must consider this an `_id`, then render it as an image. Here are the steps required:

- Remove the first slash.
- Load the corresponding document.
- Return the `photo` field in the body, assuming it is already encoded in `base64` format.
- Return a `Content-Type` header using the `photo_mime` field.

Place the following at the beginning of `main` in *control.py*:

```
if "__ow_path" in args:                                    ❶
 path = args["__ow_path"]                                  ❷
 if len(path) > 1:
   doc = model.find(path[1:])["docs"][0]                   ❸
   return {
     "body": doc["photo"],                                 ❹
     "headers": {
      "Content-Type": doc["photo_mime"]                    ❺
     }
   }
```

❶ Check if you have an extra path.

❷ Extract it.

❸ Load the image from the database.

❹ The body of the image is in the document already encoded in base64.

❺ Return the `Content-Type` header, also stored in the document.

Summary

In this chapter, we used our knowledge of Python and CouchDB to create a CRUD application in Python: a simple address book using CouchDB to store data. The application was split into two parts. In the first part, we implemented the basics of any CRUD application, reading and writing only simple string data. In the second part, we extended the application to enable pagination, validation, file upload, and rendering of images with web actions.

CHAPTER 10
Developing OpenWhisk Actions in Go

So far we have seen how to develop OpenWhisk actions in two programming languages: JavaScript and Python. Those two programming languages have many points in common: they are both interpreted languages with loosely typed variables (the common term to define them is that they are both "scripting" languages). They also have in common flexible data structures that map easily into JSON data. Probably the main difference is that most frontend developers use JavaScript, while Python is more popular among backend developers, system administrators, and even data scientists.

Given the similarities, picking one of the two is probably largely a matter of taste and available libraries. For example, JavaScript offers good libraries to manage web services, while Python has more data analysis and machine learning libraries.

Those two languages are generally productive, but the lack of strong typing can hinder development when the application grows in size, requiring you to put more effort into testing. Lack of type checking can also mean less control over data structures, which can be a problem with large programs. For those use cases, you may want to use a more strongly typed programming language.

Numerous programming languages offer stronger type checking than Python and JavaScript. The first one that comes to mind is probably Java. However, in the serverless world, the programming language on the rise is Go. Go shares many features with Java, it offers stronger type checking than scripting languages, and the "engines" powering OpenWhisk (Docker) and the preferred environment to manage Docker (Kubernetes) are both implemented in it. For those reasons, in this chapter and the next, we'll cover the implementation of OpenWhisk actions in Go.

Go has some similarities with Python and is not a complex programming language. However, it is a different breed of programming language than Python and JavaScript, mostly because you have to declare a type for each variable. This has an impact when you parse JSON data structures because you often need to know in advance the type of each field and define structures for parsing.

For obvious reasons of space and focus, I do not teach programming languages in this book. If you want to know more about the Go language, check out Go by Example.

The source code for the examples related to this chapter is available on GitHub.

Your First Golang Action

Creating an action in Go is similar to doing so in Python and JavaScript, but there are some differences related to the Go programming language. First, you start writing a function. However, in Go functions cannot be standalone but must belong to a package. By convention OpenWhisk requires you place your actions in the package `main`.

Then you need to write a function for your action. In OpenWhisk the name of the function defaults to `main`, so the natural name for an OpenWhisk function in Go would be `main`. Unfortunately, in Go, you cannot have a `main` function in the package `main`. Functions have a signature (declaring the types of their parameters), and the entry point for any Go program is `main.main()` without arguments. So, you have to consider a few Go and OpenWhisk requirements when naming your functions:

- You cannot have two functions with the same name and different parameters in Go.
- In Go the casing of the first character of a function's name defines if it is public or private (uppercase for public and lowercase for private).
- Arguments are typed. In OpenWhisk, the input and the output are JSON objects (not strings, numbers, or arrays). The type that comes closest in Go is a map whose keys are strings and whose values are unspecified (the actual type must be determined by the user by a conversion to the appropriate type). In Go this type is `map[string]interface{}`.

So, to write a Go action you have to follow these rules:

- The entry point of your Go action should be the *capitalized* version of the `main` function name.
- It must always be in the package `main`.
- It receives a parsed JSON object in the form of a `map[string]interface{}` as input.
- It returns as a result a `map[string]interface{}`.

Collecting all the rules together, the simplest possible action is an action that echoes its arguments back is (*echo.go*):

```
package main                                                    ❶
func Main(args map[string]interface{})  \                       ❷
              map[string]interface{}   {                        ❸
    return args                                                 ❹
}
```

❶ Place the action in the package `main`.

❷ The entry point is the function `Main`, receiving a parsed JSON object.

❸ The returned type is the same (note it is on the same line in the actual source code).

❹ Return the result.

Let's try to deploy the action:

```
$ wsk package update golang
ok: updated package golang
$ wsk action update golang/echo echo.go                         ❶
ok: updated action golang/echo
$ wsk action invoke golang/echo -p name Mike -r                 ❷
{
    "name": "Mike"
}
```

❶ Deploy the action in source form.

❷ A simple test.

From Echo to Hello

Now, let's write an action, *hello.go*, that does a bit more than echo the arguments. We'll also use a different name for the function `hello`.

The function:

- Reads a key from the input
- Produces a result that is a new JSON object
- Creates a new key and formats a string

Here we have to use a few more features of the Go programming language. We will:

- Add a type alias to make the code more readable.
- Convert keys because they are untyped.
- Allocate the resulting new object.
- Index members of a map to access them.

This is the code of *hello.go*:

```
package main
func Hello(args map[string]interface{}) \                    ❶
          map[string]interface{} {
   name, ok := args["name"].(string)                         ❷
   if ! ok {                                                 ❸
       name = "world"                                        ❹
   }
   res := make(map[string]interface{})                       ❺
   res["hello"] = "Hello, "+name+" !"                        ❻
   return res                                                ❼
}
```

❶ The `main` function is now `main.Hello()`.

❷ Try to get a field `name` as a string.

❸ The cast failed; there is no such field.

❹ Assign a default value for the missing field.

❺ Prepare an object to return the result.

❻ Assign the answer to the field `hello`.

❼ Return the result.

Let's deploy it and test it:

```
$ wsk action update golang/hello hello.go \
    --main hello \                                           ❶
ok: updated action golang/hello
$ wsk action invoke golang/hello -r                          ❷
{
```

```
    "hello": "Hello, world !"
}
$ wsk action invoke golang/hello \
  -p name Mike -r                                    ❸
{
    "hello": "Hello, Mike !"
}
```

❶ Note here we use a different `main` function.

❷ Test without arguments.

❸ Test with an argument.

Packaging Multiple Files

Programs written in Go, like in any other programming language, can be split into multiple files. But OpenWhisk expects a single file for deploying actions—either a source file or a zip file with multiple sources. Let's now learn the rules for packaging Go applications in zip files.

Similar to what you have already seen for JavaScript and Python, there are three cases with increasing complexity:

- Multiple files in a single package
- Multiple packages
- Multiple packages including third-party libraries

Before going into the details, let's go over how Go imports files and how the `GOPATH` works. It is important to understand how those rules apply to packaging actions.

Imports, GOPATH, and the vendor Folder

Go allows you to organize code in packages. A package is simply a string that is declared at the beginning of a source file. For example, package names like `main` and `hello` correspond to either a directory name or a relative path in the `GOPATH`, as you will see soon. As such, you can use forward slashes to nest directories.

You can compile multiple *.go* files in the same directory as long as they belong to the same package. In this simple case you do not have to set the `GOPATH`. Just `cd` into the directory and run `go build`. However, if you need to use functions belonging to other packages, you need to import them. And here is where the `GOPATH` rules apply.

As mentioned, the package name is actually a directory name. The compiler uses it to locate the directory containing the corresponding package. Keep in mind that the Go

compiler will look for packages in the subfolder *src* specified first by the `GOROOT` and then by the `GOPATH`. The `GOROOT` contains the standard libraries of the programming language and should always be set to the location where Go has been installed (if this is not set, the Go compiler is generally able to figure out where it was installed).

Let's assume you have:

- `GOPATH=/home/msciab/go`
- `GOROOT=/usr/local/go`

If you use `import "fmt"`, Go will find functions in the */usr/local/go/src/fmt* folder. If you use `import "hello"`, assuming that you have a package called `hello`, it will not find it in */usr/local/go/src/hello* so it will look in */home/msciab/go/src/hello*.

Actually, you can use fully qualified names like `github.com/sirupsen/logrus`. In this case, Go will look in */home/msciab/go/src/github.com/sirupsen/logrus*. The fact that the name looks like part of an HTTP URL is not a coincidence. It means the library is available at *https://github.com/sirupsen/logrus*. You can actually download the library in your `GOPATH` with go get `github.com/sirupsen/logrus`. The package name is generally the last part of the long name (in our case, `logrus`). However, you cannot use libraries from the `GOPATH` when sending an action to OpenWhisk. You have to use the *vendor* folder, as described next.

You can override the `GOPATH` and force a package to use a specific version of a library that you provide using the *vendor* folder. The complete search rule is this: if a package has a *vendor* folder, that folder will be searched before `GOPATH` and `GOROOT`. So let's assume you have this layout:

```
/home/msciab/go
└── src
    ├── hello
    │   ├── hello.go
    │   └── vendor
    │       └── world
    │           └── world.go
    └── world
        └── world.go
```

If in the file *hello.go* there is an `import "world"`, the Go compiler will first look in the folder *src/hello/vendor/world* and not the folder *src/world* in the `GOPATH`.

Now that the Go rules are clear, let's see how they work.

Actions with Multiple Files in main

In the simplest case, you can just split your code into different files, all in the same directory. Because you need the entry point function in the `main` package, in this case you put everything there.

To deploy, just collect the *.go* files in a zip file, without a subdirectory, and send it to OpenWhisk, *eventually* specifying the `main` function. For example, assume we have a utility function to create a `map[string]interface{}` with one field and want to place this function in the file *util.go*:

```
package main                                                    ❶
func mkMap(key string, any interface{}) \                       ❷
  map[string]interface{} {                                      ❸
        res := make(map[string]interface{})                     ❹
        res[key] = any                                          ❺
        return res
}
```

❶ Utility in the `main` package.

❷ Accept a key and a value.

❸ Return a map.

❹ Create the map.

❺ Assign the value.

Using the `mkMap` function, we can create an action `datetime` that will return the current date and time according to a format string (or throw an error if no format is provided):

```
package main                                                    ❶
import "time"
// Datetime returns date/time using a format string
func Datetime(args map[string]interface{}) \
  map[string]interface{} {
        now := time.Now()                                       ❷
        fmt, ok := args["format"].(string)                      ❸
        if ok {
                return mkMap("result", now.Format(fmt))         ❹
        }
        return mkMap("error", "no format")                      ❺
}
```

❶ This is an entry point so it must be in the package `main`.

❷ Read the current time.

❸ Extract the format string.

❹ First, use `mkMap` to return the formatted result.

❺ Second, use `mkMap` to return an error.

You can deploy a simple Go action without specifying any parameter as the runtime recognizes the *.go* extension. When using a zip file, it cannot do so from the extension so you have to add `--kind go:1.11`

Now we have an action split into two files. We can deploy it and test it as follows:

```
$ zip datetime.zip datetime.go util.go                                  ❶
  adding: datetime.go (deflated 37%)
  adding: util.go (deflated 29%)
$ wsk action update golang/datetime datetime.zip \
  --main datetime --kind go:1.11                                        ❷
ok: updated action golang/datetime
$ wsk action invoke golang/datetime \                                   ❸
   -p format "2001-01-02 3:4:5" -r
{
    "result": "18010-10-18 7:45:41"
}
$ wsk action invoke golang/datetime -r                                  ❹
{
    "error": "no format"
}
```

❶ Collect the two files in a zip file to deploy.

❷ Deploy, specifying the kind (`go:1.11`) and the `main` function.

❸ Invoke with a format string.

❹ Invoke without a format string.

Actions with Multiple Packages

Now let's assume the code is large and we need to split it into multiple packages. Here we have to follow the rules mandated by the Go programming language regarding the `GOPATH` as described in "Imports, GOPATH, and the vendor Folder" on page 261.

Let's assume we want to implement a (very simple) calculator, something that can accept an expression like `2 + 2` or `3 * 4` and return the result. We decide to place the parser in a package `parse` and the implementation of the operations in another pack-

age called `ops`. We keep the entry point function in the package `main` at the top level and reuse the utility function `mkMap` from before.

The layout of our code looks like this:

```
src
├── calc.go
├── ops
│   ├── add.go
│   └── mul.go
├── parser
│   └── parse.go
└── util.go
```

We placed our source in a *src* directory. The name *src* is mandated by Go compilation rules. When packaging the zip you will have to include only the files inside the *src* folder, not the *src* folder itself.

Actually, Go requires you to place your sources in a *src* directory because it compiles intermediate files in a sibling folder named *pkg* and stores the final executable in another sibling folder named *bin*.

This layout is required to compile code locally, and it is needed when you write unit tests. When you send a zip file to the runtime, the layout is replicated internally, but you only need to send the content of the *src* folder.

The runtime allows compiling code including packages. For simplicity, in OpenWhisk the Go runtime sets the `GOPATH` variable internally to the parent folder where the sources are uploaded. This way it allows compilation of a package named `hello` by simply storing it in the folder *hello* and importing it with `import "hello"` and then using the functions `hello.<Function>`.

Now let's go over the code of the `main` function of the calculator action. It includes the packages `ops` and `parse`. We'll omit the details of the functions defined in the packages for now, and show the rest of the code later, when discussing how to write tests for those functions.

The action logic is pretty simple. We parse the operator and the arguments with `parser.Parse()`, then we invoke the `ops.Add()` and `ops.Mul()` functions to perform the actual math:

```
package main                                                    ❶

import (                                                        ❷
        "ops"
        "parser"
)
```

```
// Main is an action that calculates an expressions
func Main(args map[string]interface{}) \
          map[string]interface{} {
        expr, ok := args["expr"].(string)
        if !ok {
                return mkMap("error", "no parameter expr")
        }
        op, a, b, err := parser.Parse(expr)                    ❸
        if err != nil {
                return mkMap("error", err.Error())
        }
        switch op {
        case "+":
                return mkMap("result", ops.Add(a, b))          ❹
        case "*":
                return mkMap("result", ops.Mul(a, b))
        default:
                return mkMap("error", "Unsupported Operation")
        }
}
```

❶ This is the entry point of the action, so it must be in the `main` package.

❷ Import packages using names corresponding to the folder names.

❸ Use the `Parse` function in the *parser/parse.go* file here.

❹ Use the `Add` function in the *ops/add.go* file.

Actions Using Third-Party Libraries

So far we've discussed how to split our code into multiple files and directories. We also used some libraries that come bundled with Go. The Go library is pretty rich and complete, but it obviously cannot cover everything—that's where third-party libraries come in handy. First we'll look at how to use those libraries in Go in general, then at how you can bundle them in your action code using the `dep` tool.

How Go Uses Third-Party Open Source Libraries

There are plenty of open source libraries for Go, and most of them are available as source code on internet, very frequently on GitHub. Among the more commonly used external libraries, there are logging, testing, error handling, cloud, and command-line support tools.

Go assumes external libraries can be downloaded and uses the internet address of the library to name it. For example, if you want to use the library `zerolog` to improve

support for log messages, you have to use the remote import path, which is also its complete internet address:

```
import "github.com/rs/zerolog/log"
```

The library is available on the internet at *https://github.com/rs/zerolog/log*, and Go assumes it can be downloaded using a revision control system like Git. To retrieve the library you usually run the command `go get github.com/rs/zerolog/log`.

Go downloads libraries under a subfolder of the *src* folder pointed to by the `GOPATH` environment variable. Generally, the `GOPATH` points to a folder *go* in your home directory. For example, if you have `GOPATH=/home/msciab/go`, then the *go get* command will download it to */home/msciab/go/src/github.com/rs/zerolog/log*. In this way, you have a unique, global collection of Go source code. The biggest problem with a global collection is that you are stuck with the specific version of each library you downloaded.

Selecting a Given Version of a Library

Generally, you do not want to just use whatever version of a library is stored in the `GOPATH`: you may want to keep specific versions of the libraries you used when you wrote and tested your code. When uploading code, you cannot include all the libraries you have in your `GOPATH`, as you may have hundreds of them. Instead, you have to use the *vendor* folder we discussed in "Imports, GOPATH, and the vendor Folder" on page 261.

You can generate the *vendor* folder and include only the libraries you are using with the `dep` tool. Let's see how. The first step is, of course, to download and install it, using the following command:

```
$ curl https://raw.githubusercontent.com/golang/dep/master/install.sh | sh
ARCH = amd64
OS = darwin
Will install into /Users/msciab/go/bin
Fetching https://github.com/golang/dep/releases/latest..
Release Tag = v0.5.0
Fetching https://github.com/golang/dep/releases/tag/v0.5.0..
Fetching https://github.com/golang/dep/releases/download/v0.5.0/dep-darwin-amd64..
Setting executable permissions.
Moving executable to /Users/msciab/go/bin/dep
```

Now, let's assume the code of our action at *src/ops/add.go*, is the following. The code is absolutely trivial but includes the library `zerolog` to generate logs in JSON format and a `log` statement using that library:

```
package ops

import (
        "github.com/rs/zerolog/log"                                    ❶
```

```
)

// Add adds 2 numbers
func Add(a, b int) int {
        log.Debug().Int("a", a).Int("b", b).Msg("Add")                    ❷
        return a + b
}
```

❶ The external library we want to use.

❷ Using the library to log the operations.

If you try to deploy this code, you will get compilation errors. This because the library is not part of the standard Go library and the runtime does not download third-party libraries automatically. The library has to be included in your zip file, so you have to download it first with `dep`, as follows:

```
$ cd src/ops
$ GOPATH=$PWD/../.. dep init
  Using ^1.9.1 as constraint for direct dep github.com/rs/zerolog
  Locking in v1.9.1 (338f9bc) for direct dep github.com/rs/zerolog
$ ls
Gopkg.lock      add.go          ops_test.go
Gopkg.toml      mul.go          vendor
$ cd ../..
```

Note that we specified `GOPATH=$PWD/../..`. This is necessary, because the runtime always places your code in its own `GOPATH` at the top level. As you can see, we have two files: *Gopkg.toml* and *Gopkg.lock*. The first two files are version files that `dep` uses to store information about the libraries included, while in the *vendor* folder there are the action libraries.

You generally store your code in a version control system. Since the *Gopkg.lock* and *Gopkg.toml* files store the information to retrieve the exact version of the library you are using (while `go get` only downloads the latest version), you usually do not store the actual *vendor* folder in your version control system, but only the two files. If you already have those files, you can use `dep ensure` to redownload the dependencies at build time.

Now you know how to retrieve the libraries you need (either with `dep init` or `dep ensure`). Once you have the *vendor* folder, you have all the code required to build your action in the runtime, so you can zip the folder and send it to OpenWhisk. Remember to change to the *src* folder before zipping to get the correct path in the zip file:

```
$ cd src
$ zip -qr ../calc.zip *
```

```
$ cd ..
$ wsk action update golang/calc calc.zip \
  --kind go:1.11
ok: updated action golang/calc
```

You already know how to deploy the action, which is actually a calculator. We expect that `calc` can evaluate simple expressions:

```
$ wsk action invoke golang/calc -p expr "2 + 2"
ok: invoked /sciabarra_cloud/golang/calc with id aff4b079e03d4102b4b079e03dd10278
$ wsk activation result aff4b079e03d4102b4b079e03dd10278
{
    "result": 4
}
$ wsk activation logs aff4b079e03d4102b4b079e03dd10278
2018-11-01T21:31:14.889100462Z stderr: \
{"level":"debug","a":2,"b":2,"time":"2018-11-01T21:31:14Z","message":"Add"}
```

As expected, there are messages in the log formatted as JSON.

Action Precompilation

You've learned how to create zip files including the source code to be sent to the action runtime. The ability of the runtime to compile code from sources for you is very valuable. But compiling code takes time, which becomes a problem when starting multiple copies of the action. When the system is under heavy load, OpenWhisk starts multiple instances of the runtime and then initializes them.

Initializing a runtime with a compilation is much slower than initializing with compiled code. OpenWhisk can suffer the problem of the "cold start": when the load increases, new instances of the runtime are created and initialized, but if the initialization takes a long time, some action invocations can also take a long time to be executed.

Using the Go runtime you can precompile the source code in binary form, generating an executable that is much faster to initialize since there is no compilation involved. Compilation happens offline before the action is sent to OpenWhisk.

You need to install Docker to be able to precompile an action using the image. In the following discussion I assume you have downloaded and installed Docker, so you have the `docker` command available.

Using precompilation, the compilation is performed by the same Docker image used to execute the code in OpenWhisk. Docker will download a local copy of the runtime, use it to perform the compilation on your computer, and then save the results locally. You can then send them to the OpenWhisk server you are using.

To precompile an action you need the same source that is normally sent to OpenWhisk, either a single file or a zip file. For example, let's assume you have a file *hello.go* with the `main` function `hello` that you normally deploy with the following:

```
$ wsk action create golang/hello hello.go --main hello
```

Then you can precompile the action with:

```
$ docker run -i \                                    ❶
  openwhisk/actionloop-golang-v1.11 \                ❷
  -compile hello \                                   ❸
  <hello.go  \                                       ❹
  >hello.zip                                         ❺
```

❶ Invoke Docker with an input.

❷ Name the Docker image.

❸ Flag to compile a function with `hello`.

❹ The input source.

❺ The output binary (zipped).

Note that:

- You invoke the runtime with `run`.
- You need the switch `-i`, as otherwise Docker cannot read the input.
- You have to use the flag `-compile` followed by the name of the `main` function to trigger compilation.
- You read the action from standard input and write the result to standard output.
- The result is also a zip, but it contains a binary, not source code.

If you unpack the resulting *main.zip* you will find only one file, *exec*, the standard for sending a zip file with precompiled binaries. Once the *main.zip* file is ready you can deploy it as an action, in the same way as the other actions:

```
$ wsk action create \
   golang/hello hello.zip \
   --main hello --kind go:1.11
```

You can also precompile source code and gather it in a zip file. In this case, you may have a zip file with sources and another with the binary executable.

Let's assume you have some sources in a folder called *src*. You may have packages and even *vendor* folders in it. Let's see how to precompile those cases:

```
$ cd src
$ zip -r ../main-src.zip *                                          ❶
$ cd ..
$ docker run -i openwhisk/actionloop-golang-v1.11 \                 ❷
  -compile main <main-src.zip >main-bin.zip
$ wsk action create golang/hello main-bin.zip                       ❸
```

❶ Collect the sources in a zip file.

❷ Invoke the runtime to use it as a compiler.

❸ Output it as a zip including an executable.

Testing Go Actions

When we used the Python and JavaScript runtimes, we had to figure out how to replicate the runtime locally. Luckily, since Golang is a compiled language, we can build our code locally with the same sources as the runtime environment.

This means we can test our actions locally using the available Go facilities. Go also includes an "example-based" testing utility that is similar to snapshot testing in JavaScript and the `doctest` feature in Python.

Writing Tests

Let's consider the admittedly trivial function `Mul` in the *ops/mul.go* file from our calculator example:

```
package ops

// Mul multiplies 2 numbers
func Mul(a, b int) int {
        return a * b
}
```

In Go, the standard practice is to test function by function. You create a file with the extension *_test.go* in the same folder as the file containing the function you want to test, then create a test function named like this function, but with the prefix `Test`. In code it is as simple as this:

```
func TestMul(t *testing.T) {                                        ❶
        if Mul(3, 2) != 6 {
                t.Fail()                                            ❷
        }
}
```

❶ The parameter `t` provides a means to communicate test results.

❷ In this case, the test is failing.

As you can see, this is pretty basic: you just use `if` and invoke `t.Fail()` when the test does not pass. What Go actually provides is a way to write and run tests but no fancy features for controlling the results.

Once you have the tests, you simply run them on the command line with `go test`. You can also easily select tests you want to run by name. For example:

```
$ cd src/ops
$ go test -v                                                    ❶
=== RUN   TestAdd
{"level":"debug","a":3,"b":2,"message":"Mul"}
--- PASS: TestAdd (0.00s)
=== RUN   TestMul
--- PASS: TestMul (0.00s)
PASS
ok _/chapter10-golang/calc/src/ops 0.009s
$ go test -run '^TestMul$'                  <2>                 ❷
PASS
ok _/chapter10-golang/calc/src/ops 0.009s
```

❶ Run all the tests in the current folder in verbose mode.

❷ Run only the tests identified by the regular expression.

Testing Using Examples

In Go, like in other languages, one of the more time-consuming activities when writing tests is coding assertions to verify the results. To save time, Go uses "example-based" testing, allowing you to verify test results simply by comparing their output with a "prerecorded" output stored in the test itself.

Let's use this technique to test the `parser.Parse()` function, whose code is as follows:

```
package parser

import (
        "fmt"
        "strconv"
        "strings"
)

// Parse parses an expression in format 'a op b'
func Parse(expr string) (string, int, int, error) {
        args := strings.Split(expr, " ")                        ❶
        if len(args) < 3 {
                return "", 0, 0, fmt.Errorf("not enough args")
        }
        a, err := strconv.Atoi(args[0])                         ❷
```

```
        if err != nil {
                return "", 0, 0, err
        }
        b, err := strconv.Atoi(args[2])                                    ❸
        if err != nil {
                return "", 0, 0, err
        }
        return args[1], a, b, nil
}
```

❶ Split `2 + 3` into an array with elements `2`, `+`, `3`.

❷ Verify that the first argument is an integer.

❸ Verify that the second argument is an integer.

Instead of complex assertions, we write a simple function to print the results (the typical function we would use when debugging) as follows:

```
package parser
import "fmt"
func print(op string, a, b int, err error) {
        if err == nil {
                fmt.Printf("%s(%d,%d)\n", op, a, b)
        } else {
                fmt.Printf("err: %s\n", err.Error())
        }
}
```

Using this function, a test "by example" for the parser can be written in a very simple and intuitive way:

```
func ExampleParse() {                                                      ❶
        print(Parse("2 + 2"))                                              ❷
        print(Parse("2 + 3"))
        print(Parse("2"))
        print(Parse("3 * a"))
        print(Parse("3 / 2"))
        // Output:                                                         ❸
        // +(2,2)
        // +(2,3)
        // err: not enough args
        // err: strconv.Atoi: parsing "a": invalid syntax
        // /(3,2)
}
```

❶ The name of an example-based test starts with `Example`.

❷ Feed some inputs to the function and print the results.

❸ The output we expect from the tests.

An example-based test is executed exactly like other tests; the only difference is that there are no assertions: the output of the tests is compared with the example output stored in the comments after the test itself.

Embedding Resources

So far, we've written simple actions invoked on the command line, without a user interface. Of course, this is not realistic. Real-world applications have user interfaces. We will use a web action for this.

As already discussed in Chapters 2 and 9, in OpenWhisk you can declare an action to be a "web action" and it will produce and handle web input and output. Let's create a web action that will generate HTML output including a style sheet, images, and even a complete JavaScript library, Vue.js. All those resources will be embedded in a single Go action that will serve all the components of our application.

In short, this is an example of a nontrivial Go actions supporting embedding resources and serving a complete, if simple, single-page application (SPA). This application has only client-based logic, but it is the foundation for more complex examples in the next chapters, which will also include server-side logic and interaction with other systems.

Using packr

Let's assume we have a SPA written in (client-side) JavaScript using the library Vue.js. We use Vue.js because it is one of the more common ones (in addition to React and AngularJS), but it also has the gift of simplicity. One of its advantages is that it can be deployed as a single file without using complex tooling.

Let's take a look at the files comprising this application. They are stored in a separate folder, a sibling of the *src* folder where we place Go sources, as follows:

```
res
├── index.html      ❶
├── style.css       ❷
├── main.js         ❸
├── logo.png        ❹
├── favicon.ico     ❺
└── vue.min.js      ❻
```

❶ HTML displaying the user interface.

❷ Stylesheet for the application.

❸ JavaScript logic of our SPA.

❹ Logo image.

❺ Website icon image.

❻ The Vue.js library, minified.

Let's now embed those files in our Go code. We use the tool `packr` for this, which creates an in-memory "box" containing the files stored in the local folder.

Follow these steps:

1. Install `packr` to transform resources in Go code.
2. Use `packr` to generate the file *src/app/a_app-packr.go* that includes the content of the files as a box.
3. Write *src/app/box.go* to extract data from the box.
4. Use the box in the rest of the code to serve resources, now embedded.

We begin by installing the library and the support tool with:

```
$ go get -u github.com/gobuffalo/packr
```

The *box.go* code is the following:

```
package app
import (
        "github.com/gobuffalo/packr"                    ❶
)
var box = packr.NewBox("../../res")                     ❷
```

❶ The library `packr` to use the embedded resources.

❷ Create a `box` object referring to the resources.

Now, if we run the code locally, it will use the actual files on our folder. The is a feature of `packr`: if the files are available on disk, they are used. If you want to embed resources in the binary, generate the Go source with this command first:

```
$ cd src
$ packr
```

This simple command will search for the function `NewBox` in the *.go* files and will generate the file *src/app/a_app-packr.go*. This file now embeds the resource files, which means the original files are no longer needed. You can compile the sources and will have a binary embedding the resources.

To deploy or precompile this application you need some additional libraries, which means you have to compile the action with *vendor* folder support. Follow the procedure described in "Actions Using Third-Party Libraries" on page 266, using `dep` to retrieve the dependencies and create the *vendor* folder.

Serving Resources with Web Actions

Now that we have all our resources embedded in the executable of the action, let's create a web action in Go that can serve the entire JavaScript action as a web page. As you probably remember, a web action is an action that can be accessed with a public URL, without authentication, returning not just JSON but also resources that a web browser can use to create a full web page. Web actions were described in "Advanced Web Actions" on page 239.

Using the web action API discussed in Chapter 9, you can implement an action that can serve static resources embedded in an SPA. Let's see how this works.

When you invoke an HTML page with a browser, it uses a URL that becomes the base for the resources embedded in the HTML. When you invoke an action it has a base URL like this:

```
https://openwhisk.bluemix.net/api/v1/web/namespace/package/webapp
```

Let's call this URL `$BASE`. When a user invokes this URL, we receive an empty variable `__ow_path`. We immediately redirect to `$BASE/index.html` (otherwise the base URL for the page would become the parent of `$BASE`, and this is not what we want). Once we've resolved the special case of the entry point, the action is invoked again, this time with `__ow_path` equal to `/index.html`. At this point, we can start to serve resources embedded in the action.

We use the "box" created before with `packr`, which contains the file *index.html*, an HTML file that has been embedded in our action. The file is then loaded by the browser, which in turn references other resources specified in the web page: stylesheet, JavaScript, images, etc. So, the browser invokes the web action again as `$BASE/style.css`, `$BASE/favico.ico`, etc.

In short, the action must search in the box for the resources specified in `__ow_path` to serve them. Unfortunately, there are two other complications:

- Each resource must be served with the proper content type.
- Some resources (images) are binaries and must be served encoded in base64.

Let's solve those problems in order. Since there is no information about the MIME type to apply to a resource, we use the extension of the filename to generate the proper content type. Most web servers work this way.

So, in the file *assets.go* we create a map from the extensions to the required content types:

```
// Content Type Map
var ctypes = map[string]string{
        "html": "text/html",
        "js":   "application/javascript",
        "css":  "text/css",
        "png":  "image/png",
        "jpg":  "image/jpeg",
        "ico":  "image/x-icon",
}
```

The next problem is distinguishing binary files from text files. This is pretty simple in our case because the only binaries in the box are the images, so we can write a function `isBinary` as follows:

```
// only images are treated as binaries
func isBinary(ctype string) bool {
        return strings.HasPrefix(ctype, "image/")
}
```

Now let's write an `Asset` function. It receives the path (`/index.html`, `/style.css`, etc.) and returns the body of the response (in base64 format for images) and the content type.

These are the steps:

1. Extract the extension to find the content type in the map.
2. Read the file from the box.
3. Encode in base64 the binaries.

In code:

```
// Asset extracts a file from the box with its content type
// Returns either a content type with "/" or an error code
func Asset(path string) (string, string) {
        // identify the content type
        splits := strings.Split(path, ".")                          ❶
        ext := splits[len(splits)-1]
        ctype, ok := ctypes[ext]                                    ❷
        if !ok {
                ctype = "application/octet-stream"                  ❸
        }
        // extract data
        var str string
        var bytes []byte
        var err error
        if isBinary(ctype) {                                        ❹
                // encode binaries in base64
                bytes, err = box.MustBytes(path)                    ❺
```

```
                if err == nil {
                   str = base64.StdEncoding.\                    ❻
             EncodeToString(bytes)
                }
        } else {
                str, err = box.MustString(path)                  ❼
        }
        if err != nil {
                return err.Error(), "404"                        ❽
        }
        return str, ctype                                        ❾
}
```

❶ Split the string on `.` to extract the extension.

❷ Read the content type from the map.

❸ Default to this content type if not found.

❹ Check if the content type refers to a binary.

❺ Extract from the box the path as an array of bytes.

❻ Encode the bytes as a base64 string.

❼ It was not a binary, so extract as a string.

❽ If not found, return an error code instead of a content type.

❾ Final result—return the body and content type.

Now that we have the ability to extract from the box the data with the right content type, we can use a web action to return a proper answer. We need a response with three values:

- The `body` of the response
- The `statusCode` (generally `200` except in case of errors)
- The `Content-Type` header

Here is the code:

```
// WebResponse returns a full response
// suitable for a web action
func WebResponse(path string) map[string]interface{} {
        // interpret as an asset
        body, ctype := Asset(path)                               ❶
        // prepare the answer
        res := make(map[string]interface{})
```

```
        res["body"] = body                                              ❷
        if strings.Index(ctype, "/") != -1 {                            ❸
                // asset found
                res["headers"] = map[string]string{
                        "Content-Type": ctype,                          ❹
                }
                res["statusCode"] = "200"                               ❺
        } else {
                // asset not found
                res["statusCode"] = ctype                               ❻
                res["headers"] = map[string]string{}
        }
        return res
}
```

❶ Retrieve the asset.

❷ Set the body of the answer.

❸ Detect if the answer is an error.

❹ Set the Content Type header.

❺ Status code is `200` when the answer is OK.

❻ Set the status code when there is an error.

We are now ready to write the `main` action. It returns the result of the `WebResponse` function. The only special case is returning a redirection when the path is empty. Since we are going to serve a JavaScript app, we need to have JavaScript enabled, hence we're using a JavaScript redirection.

The entry point action just needs to read the path and generate a redirect if empty. Otherwise, we invoke `WebResponse` and return the results:

```
// Main is the main action
func Main(args map[string]interface{}) \
         map[string]interface{} {
        // get the path
        path, ok := args["__ow_path"].(string)                          ❶
        if ok && path != "" {
                return app.WebResponse(path)                            ❷
        }
        return map[string]interface{}{
                "body": `                                               ❸
        <script>
          location.href += "/index.html"
        </script>
                `,
```

```
        }
}
```

❶ Read the path.

❷ Expand it with `WebReponse` if not empty.

❸ Return a JavaScript snippet for the redirection.

Accessing the OpenWhisk API in Go

We'll complete the chapter by wotking through a detailed example of how to access the OpenWhisk REST API using direct `HTTP` requests. In particular, you will see:

- How to construct a web request to invoke the OpenWhisk REST API
- An example of action invocation
- An example of firing a trigger
- How to retrieve data with an activation id
- How to test functions including API calls

We covered the REST API in detail in "Using the OpenWhisk REST API" on page 162. Now we're going to invoke the API using Go. First we'll construct the function s `url` and `auth`, to build an OpenWhisk URL. Then we'll introduce a couple of utility functions for creating requests and responses: `mkMap` and `mkErr`. Next, we'll explore how you create HTTP requests in Go, with `mkPost` and `doCall`. Finally, we'll implement the `whiskInvoke` function to invoke an OpenWhisk action. The last step is to deploy an example `Invoke` function that invokes the utility function `sort` from the `utils` library.

Let's get started.

Utilities

The format of REST API URLs is:

```
https://{APIHOST}/api/v1/namespaces/{NAMESPACE}/{ENTITY}/...
```

When creating the URL to access OpenWhisk, the required information is either stored in environment variables or passed as a parameter. Hence, we can build a `url` function to easily generate those URLs as follows:

```
func url(entity string) string {
        return fmt.Sprintf("%s/api/v1/namespaces/%s/%s",                    ❶
                os.Getenv("__OW_API_HOST"),                                 ❷
                os.Getenv("__OW_NAMESPACE"),
```

```
                entity)
}
```

❶ Use a format string to create the URL.

❷ Extract parameters from the environment variables.

The invocation of a URL with a username and password requires separate values for each. Since those two values are passed as a single string in the environment, we need a function to split them:

```
func auth() (string, string) {
    key := os.Getenv("__OW_API_KEY")                    ❶
        up := strings.Split(key, ":")                   ❷
        return up[0], up[1]                             ❸
}
```

❶ Extract the API key from the environment.

❷ Split the string into an array of two components.

❸ Return the values as multiple return values.

A common need in the implementation that follows is returning a map from one single value. Since creating one is a bit verbose, it makes sense to create a `mkMap` function to simplify this common task:

```
func mkMap(key string, value interface{}) map[string]interface{} {
        return map[string]interface{}{
                key: value,
        }
}
```

In OpenWhisk, errors are returned as JSON objects with a string field named `error`. But you may want to create an error in different cases: when you receive an `error` from a function, when you want to generate an error message, or when you want to return the content of a Go data structure to help to debug the error. So, we'll also write a `mkErr` function using the type switch feature of Go: the generated error is based on the type of the value. In code:

```
func mkErr(err interface{}) map[string]interface{} {
        switch v := err.(type) {                        ❶
        case error:
                return mkMap("error", v.Error())        ❷
        case string:
                return mkMap("error", v)                ❸
        default:
                return mkMap("error",                   ❹
          fmt.Sprintf("%v", err))
```

```
        }
}
```

❶ Dispatch the type of the argument.

❷ Get the error string if you have an actual error.

❸ Send as is if you have a string.

❹ In other cases, just return the representation of the data structure using the Go function `fmt.Sprintf()`.

HTTP Requests

To make an HTTP request in Go, you need to construct a pretty complex data structure and then use a client to actually perform the request. Since this is more complex than in Python, we'll split this request into two different functions: `mkPost` to prepare the request and `doCall` to actually execute it.

The `mkPost` construct is an object of type `*http.Request`. Since we execute JSON requests and expect JSON responses, our `mkPost` will take care of converting Go data structures into JSON format for sending. Furthermore, the function `doCall` will perform the conversion back from JSON to Go data structures.

Getting into the code, keep in mind that:

- `mkPost` expects the entity name and the arguments in `map[string]interface{}` format.
- It uses `url` to generate the complete URL and `auth` to provide authentication information.
- Encoding from Go types to JSON is performed by `json.Marshal()`.
- The request must specify it is sending JSON content with its `MIME` type.

It looks like this:

```
func mkPost(entity string,
        args map[string]interface{}) \                          ❶
            (*http.Request, error) {
        data, err := json.Marshal(args)                         ❷
        if err != nil {
                return nil, err
        }
        req, err := http.NewRequest("POST", url(entity),        ❸
                bytes.NewBuffer(data))
        if err != nil {
                return nil, err
```

```
        }
        req.SetBasicAuth(auth())                                    ❹
        req.Header.Set("Content-Type", "application/json")          ❺
        return req, nil
}
```

❶ The function returns either an HTTP request or an error.

❷ Encode the arguments in JSON .

❸ Create a `POST` request with the URL and the JSON payload.

❹ Add authentication information.

❺ Set the `Content-Type` to the JSON MIME type.

Now we can write the `doCall` function which, given a request, performs the actual HTTP call. Note the function is generic: it also works for other kinds of requests (e.g., `GET`), and we will reuse it later.

Here are a few key points to remember:

- You need to create a client to perform the actual request.
- The answer is a `Reader` and must be read entirely to get the body.
- The function expects a JSON object and performs the decoding.

The code is as follows:

```
func doCall(req *http.Request) \
     map[string]interface{} {
        client := &http.Client{}                                    ❶
        res, err := client.Do(req)                                  ❷
        if err != nil {
                return mkErr(err)
        }
        body, err := ioutil.ReadAll(res.Body)                       ❸
        if err != nil {
                return mkErr(err)
        }
        // encode answer
        var objmap map[string]interface{}
        err = json.Unmarshal(body, &objmap)                         ❹
        if err != nil {
                return mkErr(err)
        }
        return objmap
}
```

❶ Create an HTTP client.

❷ Perform the actual call.

❸ Read the body in a byte array.

❹ Decode the byte array in a map.

Invoking an OpenWhisk Action

The `whiskInvoke` function invokes an OpenWhisk action with its payload. The action invocation can be either blocking or not and can return the result of the invocation or just data about the invocation. Hence, it requires two flags as parameters, in addition to the action name and payload.

When we reuse the functions we've already developed, the code is pretty straightforward:

```
func whiskInvoke(action string, \
   args map[string]interface{},
        blocking bool, result bool) \
      map[string]interface{} {
        invoke := fmt.Sprintf(
     "actions/%s?blocking=%t&result=%t",                    ❶
                action, blocking, result)
        req, err := mkPost(invoke, args)                    ❷
        if err != nil {
                return mkErr(err)
        }
        return doCall(req)                                  ❸
}
```

❶ Build the request, including options for blocking and returning results.

❷ Create the `POST` request from the parameters.

❸ Execute the `POST` request.

Let's test it, invoking a `sort` action. The code of this example is longer because there is a good amount of checking for missing parameters, but the logic is pretty simple. The code:

- Receives an action to invoke and a message to print
- Gets a text to sort, as a comma-separated string
- Splits the string into a list of lines and sends it to the `sort` action
- Invokes the action
- Collects the result in a single string to be returned

Here it is:

```
// Invoke invokes the sort using the action parameter specified
func Invoke(args map[string]interface{}) \
   map[string]interface{} {
        // retrieve action
        action, ok := args["action"].(string)                    ❶
        if !ok {
                return mkErr("no action")
        }
        // prepare args
        text, ok := args["text"].(string)                        ❷
        if !ok {
                return mkErr("no text")
        }
        input := strings.Split(text, ",")                        ❸
        // invoke action
        res := whiskInvoke(action,                               ❹
      mkMap("lines", input), true, true)
        log.Printf("%v", res)
        lines, ok := res["lines"].([]interface{})
        if !ok {
                return mkErr("cannot retrieve result")
        }
        // retrieve message
        result, ok := args["message"].(string)
        if !ok {
                result = ">>>"
        }
        for _, v := range lines {                                ❺
                result += " " + v.(string)
        }
        return mkMap("result", result)
}
```

❶ Get the action name to invoke, or an error.

❷ Get a message to print, or an error.

❸ Split the text into an array.

❹ Invoke the action.

❺ Concatenate the message and snippets in a single line.

Firing a Trigger

Now we are going to see how to fire a trigger, using the trigger API. It is always asynchronous and returns an activation id. To see the result you also have to retrieve the result using the activation API.

To use the code, you need to create a trigger and some rules. Let's create a trigger and then attach one rule, invoking the `sort` action:

```
$ wsk trigger create golang-trigger                              ❶
ok: created trigger golang-trigger
$ wsk rule update golang-trigger-sort \                          ❷
     golang-trigger utils2/sort
ok: updated rule golang-trigger-sort
$ wsk rule enable golang-trigger-sort                            ❸
ok: enabled rule golang-trigger-sort
```

❶ Create a trigger.

❷ Create a rule invoking `sort` on the trigger.

❸ Enable the rule.

Now we are ready to write a function that fires the trigger.

The function `whiskTrigger` is not very different from the `whiskInvoke` function we wrote earlier. But it is simpler because you do not need to specify a blocking behavior, since it is always nonblocking. The function is as follows:

```
func whiskTrigger(trigger string,
        args map[string]interface{}) \
      map[string]interface{} {
        invoke := fmt.Sprintf("triggers/%s", trigger)            ❶
        req, err := mkPost(invoke, args)
        if err != nil {
                return mkErr(err)
        }
        return doCall(req)
}
```

❶ Use the REST call for `triggers`.

Note that firing a trigger returns an activation id. You can see this on the command line (refer to "Using the OpenWhisk REST API" on page 162 for details about how to configure environment variables):

```
$ curl -su $AUTH \
  -X POST $URL/triggers/golang-trigger \                         ❶
  -H "Content-Type: application/json" \
  -d '{"lines":["b","a","c"]}'                                   ❷
{"activationId":"2275e8aa39a743c4b5e8aa39a7a3c44a"}              ❸
```

❶ URL to invoke the trigger posting a JSON object.

❷ Send data to sort as an array of lines.

❸ The answer is an activation id.

So what do we do next? Of course, we retrieve the data associated with the activation id.

Retrieving the Data Associated with the Activation ID

The function `whiskRetrieve` is similar to those we wrote before. The main difference is it is using a `GET` request, so we need a `mkGet` first. It is very similar to `mkPost`, only simpler since there is not a body:

```
func mkGet(action string) \
        (*http.Request, error) {
        req, err := http.NewRequest("GET", \                    ❶
       url(action), nil)
        if err != nil {
                return nil, err
        }
        req.SetBasicAuth(auth())                                ❷
        return req, nil
}
```

❶ Create a `GET` request.

❷ Set authentication information.

The function `whiskRetrieve` just performs a `GET` request with the activation id:

```
func whiskRetrieve(id string) \
       map[string]interface{} {
        invoke := fmt.Sprintf(
        "activations/%s", id)                                   ❶
        req, err := mkGet(invoke)                               ❷
        if err != nil {
                return mkErr(err)
        }
        return doCall(req)
}
```

❶ Build an invocation with the activation id.

❷ Construct a `GET` request.

Summary

In this chapter, you learned how to write OpenWhisk actions using Go. First, we covered the peculiarities of the OpenWhisk Go runtime and its way of creating actions and running tests. Then we discussed advanced features, like how to precompile actions, how to use vendoring to include third-party libraries, and how to use resour-

ces in a Go action with the `packr` tool. Finally, we explored how to connect to the OpenWhisk API using the REST interface, writing code to invoke other actions, fire triggers, and retrieve results.

CHAPTER 11
Using Kafka with OpenWhisk

OpenWhisk can be considered an event-based system in which some event sources produce events that are processed by actions. So far we've looked at event sources like HTTP requests, handled by web actions and direct action invocations originated by other actions.

Those are only two of many possible event sources. The strength of OpenWhisk is that you can plug in as event sources other internet-based services. For example, an event source can be Cloudant notifying of database changes, GitHub of notifying commits; or Slack announcing messages. Packages for managing those sources are included with OpenWhisk.

Event-based systems are frequently built using Apache Kafka. Kafka is a popular solution for event processing because it has many desirable properties for building event processing systems. First, it is *scalable* and *distributed*, so it can grow to handle millions of events per second. You deploy Kafka in clusters, and you can add more servers to spread the load among them. Second, it is *durable*, in the sense that notifications of events (also called *messages*) are not thrown away but can be saved for later reuse. Third, it is *replicated* and *partitioned*, so messages are not stored only on one server; there are copies in different servers.

OpenWhisk provides a `/whisk.system/messaging` package to manage messages. It is a frontend package for using Kafka, similar to the `/whisk.system/cloudant` package, a frontend to CouchDB and Cloudant.

In this chapter, you will learn how to use the `messaging` package and hook it into triggers and feeds. However, since this package does not allow us to leverage the full potential of Kafka, we'll also demonstrate how to use Kafka directly with Kafka libraries to send and receive messages. We'll use Go and the `confluent-kafka-go` library (supported by the OpenWhisk Go image) for the examples. Finally, we'll put the

examples to work by writing a web chat application in JavaScript (hosted in a Go action) that uses Kafka and the sender and receiver actions that we wrote for Go in the previous chapter.

But before starting, we need an actual Kafka instance. We'll begin by provisioning Kafka cluster in the IBM Cloud.

Using OpenWhisk and Kafka can be challenging. OpenWhisk by design scales automatically, creating new instances of action executors when there are more requests. However, if there are too many requests trying to connect to the Kafka brokers, Kafka starts to refuse new connections, causing applications to fail. With Kafka, you are required to carefully size the Kafka cluster to the expected load to avoid this problem, as it is not handled automatically by OpenWhisk. Kafka may not scale when a large number of connection requests are generated. There is work in progress to solve this and similar problems through the use of concurrency for the action runtimes.

The source code for the examples related to this chapter is available on GitHub.

Introducing Apache Kafka

Apache Kafka is, at its core, a reliable and scalable streaming platform. A high-level overview of its architecture is presented in Figure 11-1.

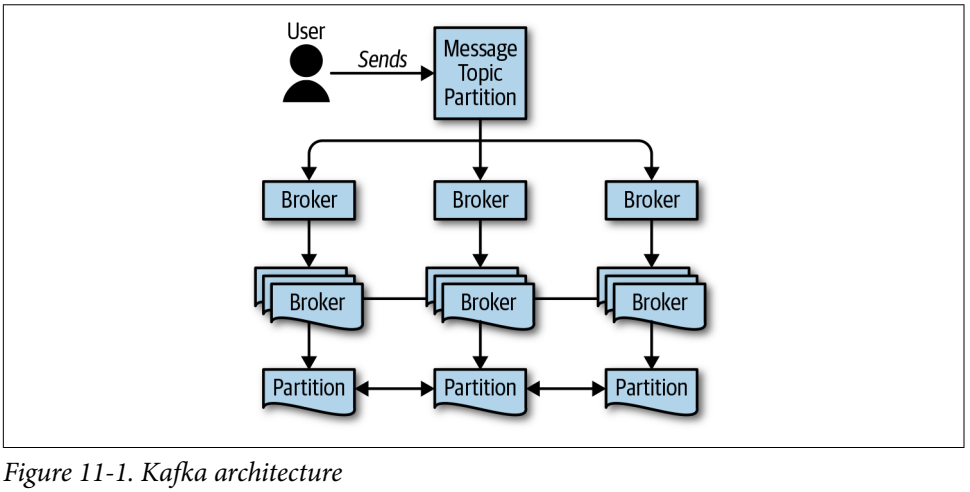


Figure 11-1. Kafka architecture

When deployed, there is a cluster of servers that a potentially large number of clients can access. Clients either produce (send) or consume (receive) messages. To achieve scalability, it distributes the workload over the servers. To achieve reliability, it replicates data in different servers.

To better understand Kafka you need to understand:

- Brokers and the protocol
- Messages and keys
- Topics and partitions
- Client groups

Kafka Brokers and Protocol

Strictly speaking, when we talk of Apache Kafka, we're referring to a cluster of Kafka servers, called *brokers*. Most developers use multiple brokers, because the ability to scale is essential for reliability and performance. One key feature of Kafka is the ability to add and remove servers while keeping the cluster running. However, the servers alone are useless without clients. Clients are applications connecting to brokers using the Kafka protocol.

It is important to note that the Kafka protocol is not HTTP, nor a variant of it. It is a Kafka-specific protocol optimized for streaming and designed for long-running connections. Applications only need to speak its protocol to use Kafka. In practice, ready-to-use libraries implementing it produce or consume messages in the more widely used programming languages. In many cases, there are multiple libraries for the same programming language.

Kafka itself is implemented in Scala, runs on the Java VM, and includes a Java library developed together with the servers as the primary implementation of the Kafka protocol. Besides the Java library, the reference implementation, another non-Java implementation of the protocol is the C library called `librdkafka`. Since many programming languages can use C libraries, this library is the basis for lots of other implementations, such as the command-line client `kafkacat` and Kafka libraries in Python and Go.

In this chapter, we'll see examples in Go using the library `confluent-go-kafka` built on top of `librdkafka`. The runtime of `librdkafka` is included in the Go runtime for OpenWhisk to simplify the use of the Go libraries that require it.

To connect to a Kafka cluster you need to specify the IP address and the port of one of the brokers, called the *bootstrap* server. Once connected to one broker the protocol communicates the locations of the other brokers.

Since any brokers can go offline, to avoid a single point of failure, more than one broker should be provided to bootstrap access to the cluster.

Access to a cluster can be open to everyone or it can require authentication. Furthermore, data can be sent to the broker in plain text or can be encrypted (e.g., with the widely used encryption protocol TLS).

Messages and Keys

Kafka clients produce messages. A message has a *key*, a *body*, and a *timestamp*. The timestamp records the time the producer sent a message and lets Kafka order the messages by production time. The body of a message is just an arbitrary sequence of bytes that can be interpreted by the consumer as it likes (e.g., as a JSON object, as a string, or as an image). Keys are also sequences of bytes, and they can also be empty. Keys are important to distribute the load, since partitions use them to select where to place the data.

Topics and Partitions

Producers can send messages to consumers, but how can they distinguish them? Not all the messages are equals, and they may have different purposes: the notification for a user login is different from logging an error.

Kafka stores messages in *topics*. Every message must specify the topic, and messages produced for a topic can be read only by clients that subscribe to that topic.

A topic can be created automatically at the first use, or an administrator can create it when needed. Note that for each topic you need to specify a *retention period*: an hour, a day, a week, or forever. Kafka discards messages from a topic after this retention period.

A topic is not monolithic. To achieve scalability and distribute the load among various servers, Kafka splits topics into *partitions*. Partitions are identified by integer numbers.

When sending a message, a producer must specify both the topic and the partition. To select a partition, a client can specify the partition to use, use a round-robin algorithm, or, if it has a key, use a hash algorithm. In general, the purpose of a key is to select the partition in which to store a message.

Offsets and Client Groups

Messages sent to Kafka belong to a partition of a topic. Within a partition, messages are ordered strictly in the order they arrive. Consumers can read the messages of the specific partition within the topic they subscribe to. But the question is: starting from which message? All the messages still available within the retention period, or all the messages starting from the moment the consumer started to read them? And what happens to messages it is already read?

The answer is simple: Kafka classifies consumers in *consumer groups* and only keeps the offset of the last message that a consumer read. When a user creates a consumer group, it requests to initialize the offset to some value (usually the last message available in the queue at creation time); then for each message consumed the offset is incremented.

So when you want to consume messages, you have to specify the topic to subscribe to, your consumer group, and the initial offset. Then the offset is automatically incremented at each read. You can also select the partition to use or let Kafka assign a partition for you.

Each consumer in the consumer group will read a message in the topic and partition it subscribed to once, starting from the initial offset they chose.

If you have one consumer in a consumer group, it will see all the messages; if you have more consumers in the same group, only one of the group will see each message, once.

Creating a Kafka Instance in the IBM Cloud

Our first step, before actually using the `messaging` package, is to provision a Kafka instance, then get the credentials to access it. Because we're using the IBM Cloud, the most straightforward route is to create one there. In the IBM Cloud, messages queues are called *event streams*, but they are actually Kafka brokers we can use for our purposes.

Event stream instances in the IBM Cloud are not free. If you create one, you will be charged. But a single partition for simple usage is not expensive.

I assume you already have an account in the IBM Cloud and will provision a Kafka instance to use as the backend of our messaging examples.

To use an instance you need to:

1. Create a broker in the IBM Cloud.
2. Select the broker and then create a topic in it.
3. Retrieve the credentials to access the broker.

Creating an Instance

Figure 11-2 shows the steps required to create an instance of the Kafka service in the IBM Cloud. Note that when you register, you are placed in a namespace and you have to pick a region to use.

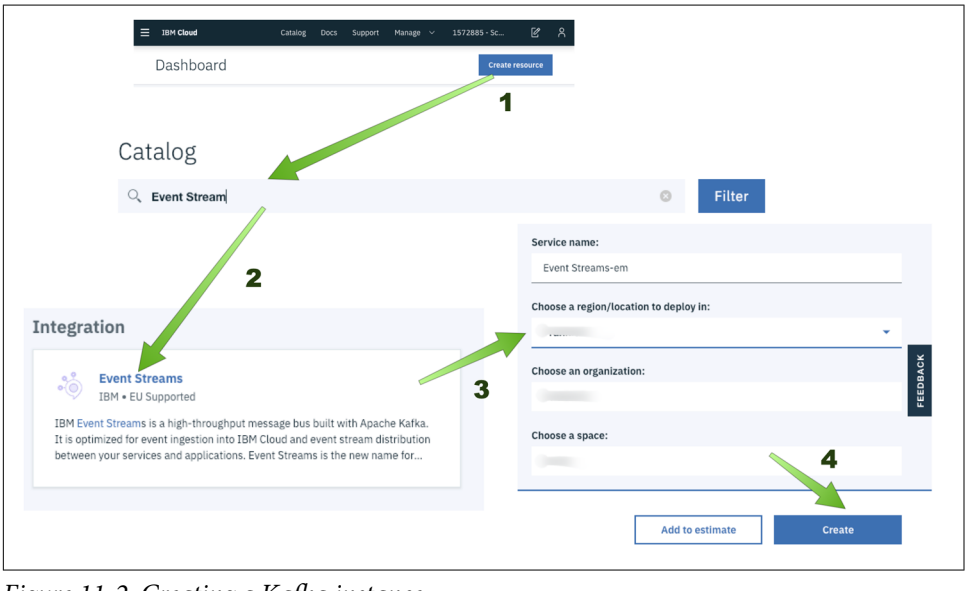


Figure 11-2. Creating a Kafka instance

Here are the steps, as outlined in Figure 11-2:

1. Log into the IBM Cloud, go to the Dashboard by clicking the IBM Cloud link, and then click "Create resource."
2. In the Catalog, search for "Event Stream" and press Enter. The catalog will show the Event Streams resource.
3. Click the Event Streams link; it will bring you to the creation form.
4. Fill in the parameters, selecting your organization, the region, and finally a namespace; then click Create.

Creating a Topic

Now that you have a new instance, before you can use it, you need to create a "topic," which will be our message queue. Figure 11-3 shows how to do that.

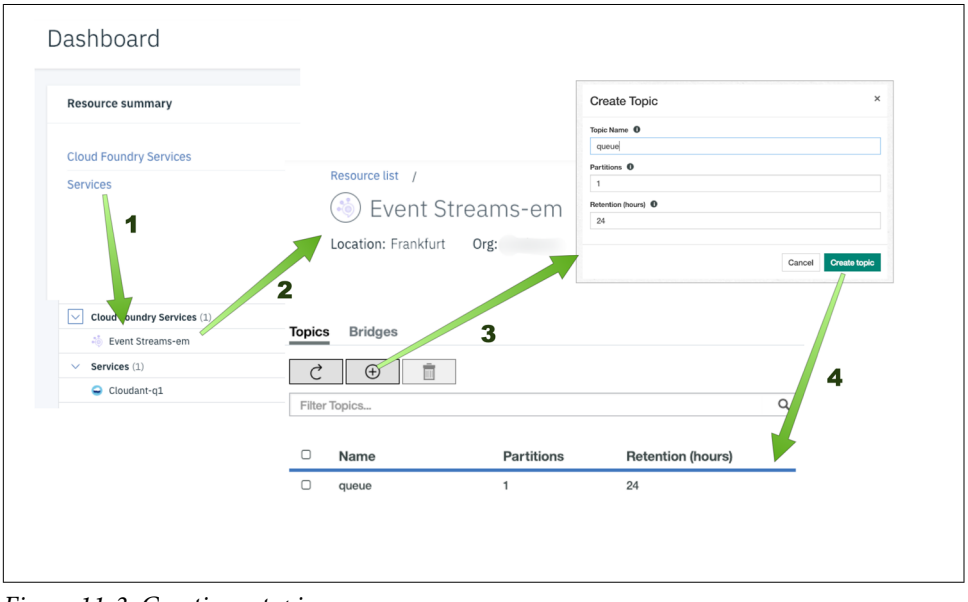


Figure 11-3. Creating a topic

Here are the steps outlined in Figure 11-3:

1. Starting again from the Dashboard, click Services and expand Cloud Foundry Services; you will see a link to your newly created broker.
2. Click that link to reach a page to manage the topics.
3. Click the + button in the Topics list to open a pop-up form to create a topic.
4. Give your topic a name (e.g., "queue"), accept the defaults of 1 partition and 24 hours retention, and click "Create topics."

Now you'll see the newly created topic in the list.

In the IBM Cloud, the price of the MessageHub depends on the number of partitions.

Get Credentials

We have a broker with a topic, but to use it we need to get the credentials. Those are available in a convenient JSON format that can be fed to OpenWhisk actions. Figure 11-4 shows the steps to get the credentials.

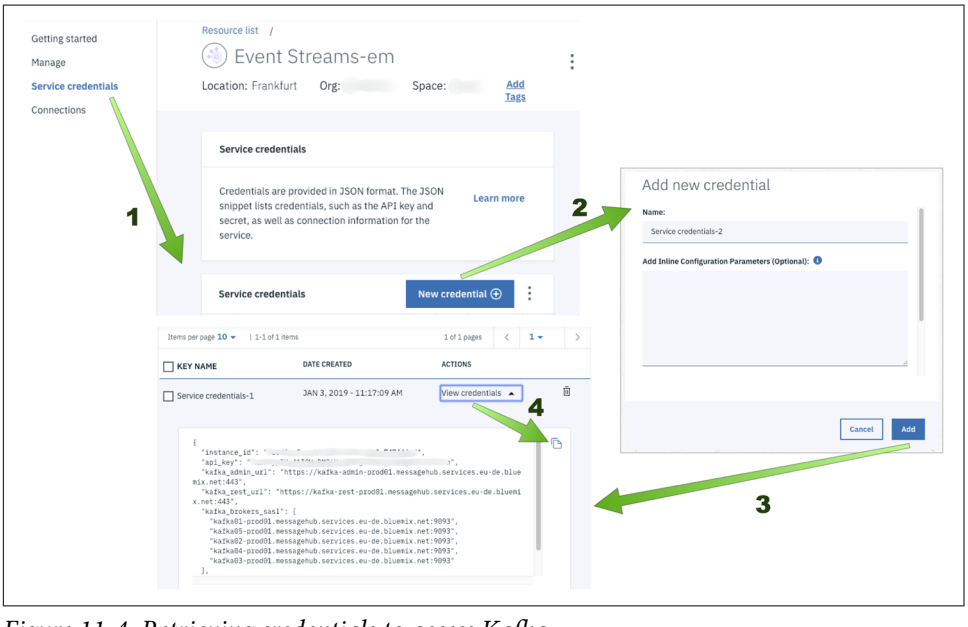


Figure 11-4. Retrieving credentials to access Kafka

You need to perform these steps:

1. Starting from the Broker page you reached in the previous step, click "Service credentials."
2. Click the "New credential" button to open a form for creating credentials.
3. Click Add to accept the defaults.
4. Now you have a new credential set; you can see it by clicking "View credentials," which will expand to a text area showing a JSON document with the parameters you need.

You can now copy the JSON document with the credentials to the clipboard and save them as a local file called *cred.json*.

Do not forget to create the *cred.json* file. We are going to use it in the coming examples to create an instance of the `messaging` package to access the Kafka cluster we have created.

Using the messaging Package

Now that we have a broker and a topic, we can begin to produce and consume messages. Here we'll use the standard package, `/whisk.system/messaging`. This package has a feed to notify of messages received in a topic, so you can hook an action to consume them. The package also includes a producer action, but it is deprecated and might disappear at any time, so we don't use it. Instead, to send messages, we'll use the command-line tool `kafkacat`. Later in the chapter, I'll show how to produce and consume messages in actions using a Go library that talks directly to Kafka.

Let's focus on the `messaging` package. To use it, we'll need to follow these steps:

1. Bind the `messaging` package to our package so you can use the broker we created.
2. Create a trigger that hooks into a feed from the bound package.
3. Create an action to display the received messages.
4. Add a rule to invoke the action.
5. Test that this chain works with `kafkacat`.

We discuss each of these steps in the following sections.

Creating a Binding and a Feed

In the section "Creating a Kafka Instance in the IBM Cloud" on page 293, we created a Kafka cluster and then downloaded the file *cred.json* with the credentials and the other parameters needed to access that cluster. We are now going to use that file as a parameter file for the binding. With the following command we can create the package `demoq` that can access the provisioned Kafka cluster:

```
$ wsk package bind /whisk.system/messaging \
  demoq -P cred.json
ok: created binding demoq
```

To use this package and receive messages from it, let's create a trigger using the action `messageHubFeed` from the bound package as a feed. We also need to specify the parameter `topic` to correctly select the source of the messages:

```
$ wsk trigger create messages-trigger \
        -f demoq/messageHubFeed \                    ❶
        -p topic queue \                             ❷
```

```
        -p isJSONData true                                              ❸
ok: created trigger messages-trigger
```

❶ Use the package binding `demoq` to connect to our broker.

❷ Specify the topic we created.

❸ Messages should be parsed as JSON data.

Now when a producer delivers a message in the topic `queue`, it will fire the trigger. But the trigger is not enough—we need an action to receive messages.

Receiving Messages with an Action

To see the messages, we need to write a simple action in Go. The purpose of this example is to investigate what the package does , so we'll use a simple action that just prints the received messages:

```
package main

import "log"
import "encoding/json"

// Main is a logger function
func Main(args map[string]interface{}) \
          map[string]interface{} {
        obj, _ := json.Marshal(args)                                    ❶
        log.Printf("%s\n", obj)                                         ❷
        return args
}
```

❶ Encode the arguments in JSON.

❷ Write to standard error.

Now you can deploy the action and hook it into the trigger with a rule:

```
$ wsk action update mesg/logme logme.go                                 ❶
ok: updated action mesg/logme
$ wsk rule update messages-trigger-logme \                              ❷
messages-trigger mesg/logme
ok: updated rule messages-trigger-logme
$ wsk rule enable messages-trigger-logme                                ❸
ok: enabled rule messages-trigger-logme
```

❶ Deploy the `logme` action.

❷ Add a rule to trigger the action when a message arrives.

❸ Enable the rule.

Sending Messages Using kafkacat

To send messages we are going to use the utility `kafkacat`, available on macOS or Ubuntu-based Linux systems. You can install it with `brew install kafkacat` on macOS or `apt-get install kafkacat` on Ubuntu.

In order to use it with a broker using the Secure Sockets Layer (SSL) encryption protocol, as described by the *cred.json* file, you need to extract some values from it and pass a number of parameters. Since using it on the command line is unpractical, we use the script *kat.sh* to launch it as follows:

```
#!/bin/bash
BROKERS="$(jq -r '.kafka_brokers_sasl|join(",")' <cred.json)"
USER="$(jq -r .user <cred.json)"
PASS="$(jq -r .password <cred.json)"
kafkacat \
  -b "$BROKERS" \
  -X sasl.username="$USER" \
  -X sasl.password="$PASS" \
  -X sasl.mechanisms=PLAIN \
  -X security.protocol=sasl_ssl \
  "$@"
```

If you do not have a macOS or Ubuntu system you can use `kafkacat` on other systems, like Windows, using Docker. In the repository there is a folder *kat* with a Dockerfile that has been tested on Windows. You need to:

- Copy your *cred.json* into the *kat* folder.
- Build a Docker image with `docker build -t kat`.

Now you can use the same commands, replacing `./kat.sh` with `docker run -i kat`.

Testing the Kafka Broker

As a first step, check if you can connect to your cluster with our script, listing the available brokers and topics with the `-L` option. You should see output like this:

```
$ ./kat.sh -L
Metadata for all topics (from broker 1: \
sasl_ssl://kafka02-prod01.messagehub.services.eu-de.bluemix.net:9093/1):
 5 brokers:
  broker 2 at kafka03-prod01.messagehub.services.eu-de.bluemix.net:9093
  broker 4 at kafka05-prod01.messagehub.services.eu-de.bluemix.net:9093
  broker 1 at kafka02-prod01.messagehub.services.eu-de.bluemix.net:9093
  broker 3 at kafka04-prod01.messagehub.services.eu-de.bluemix.net:9093
  broker 0 at kafka01-prod01.messagehub.services.eu-de.bluemix.net:9093
1 topics:
```

```
  topic "queue" with 2 partitions:
    partition 0, leader 3, replicas: 3,0,2, isrs: 3,0,2
    partition 1, leader 0, replicas: 0,2,4, isrs: 0,2,4
```

Now, to visualize what our action receives, enable polling with:

```
$ wsk activation poll mesg/logme.
```

In another terminal window, try to send messages. To do so, you should use the `-P` flag (for "produce") and specify the topic with `-t`. Messages are in standard input, so you feed those messages with a pipe. Also, note that you do not have to specify a key, but if you want to, you can specify it as a prefix in the message, and specify the separator character with `-K`. For example, in the other terminal:

```
$ echo test | ./kat.sh -P -t queue                                    ❶
Activation: 'logme' (e6b788dc35364a23b788dc35366a23bf)
[
    "2019-01-27T12:04:26.850048794Z stderr: 2019/01/27 12:04:26 \      ❷
    {\"messages\":[{\"key\":null,\"offset\":0,\"partition\":1,\
     \"topic\":\"queue\",\"value\":\"\\\"test\\\"\"}]}"
]
$ echo hello:world | ./kat.sh -P -t queue -p 0 -K:                     ❸
Activation: 'logme' (5c6e09b49cc5453aae09b49cc5653abc)
[
    "2019-01-27T12:08:41.721368293Z stderr: 2019/01/27 12:08:41 \      ❹
    {\"messages\":[{\"key\":\"hello\",\"offset\":1,\"partition\":0,\
    \"topic\":\"queue\",\"value\":\"\\\"world\\\"\"}]}"
]
```

❶ Send a simple message with no key and no partition.

❷ In the other terminal: First message (offset 0) received in topic `queue` partition 1, key `null`, value `test`.

❸ Send a message with a key (separator `:`) in partition 0.

❹ In the other terminal: Second message (offset 1) received in topic `queue` partition 0, key `hello`, value `world`.

You can also use *kat.sh* as a consumer. Use `-C` for consumer mode, and you can use `-o beginning` to specify the offset and read messages from the beginning. You can also use `-f` to specify a format string (just type `./kat.sh` to see the available options). For example, to dump the entire queue:

```
$ ./kat.sh -C -t queue -f '%k:%s\n' -o beginning
:test
hello:world
% Reached end of topic queue [1] at offset 1
% Reached end of topic queue [0] at offset 1
```

A Kafka Producer in Go

Earlier, we used Kafka with a command-line client. This is useful for testing and debugging, but you may want to be able to send messages to a Kafka broker on your own, and you can do this by connecting directly to a Kafka broker with a Kafka client library.

In this section you'll learn how to write such an action in Go, using the library `github.com/confluentinc/confluent-kafka-go`, an excellent Go library based on the C library `librdkafka` (the same one used by the `kafkacat` tool). Since connecting to a broker has a cost, we try to minimize the impact by caching the Kafka connection in the action. But before we get started, be sure you've read the warnings at the beginning of this chapter carefully.

Creating a Producer

To use the library, the first step is of course to include it. We are going to put everything in the `main` package because this is a simple enough action. Furthermore, we need to log errors, so we'll start by writing a *main/send.go* file with this code:

```
package main

import (
        "log"

        "github.com/confluentinc/confluent-kafka-go/kafka"
)
```

Now, we need to create a producer. There is a `kafka.NewProducer()` method, but it requires a pretty complex configuration in the form of a `ConfigMap`, so we'll first write a `configProducer` function that uses the configuration parameters from the *cred.json* file you created when setting up the Kafka cluster. Note that here we assume you are creating the action with `-P cred.json`, feeding the parameters generated by the Kafka instance you provisioned in "Creating a Kafka Instance in the IBM Cloud" on page 293.

The parameters are similar to those you used to connect to the cluster with `kafkacat`. You have to specify the Kafka server to connect to, then pick `sasl` as the security mechanism and provide a username and password:

```
func configProducer(args map[string]interface{}) *kafka.ConfigMap {

        // extract broker list from map
        brokers := ""
        for _, s := range args["kafka_brokers_sasl"].([]interface{}) {          ❶
                brokers += s.(string) + ","
        }
        brokers = brokers[0 : len(brokers)-1]
```

```
        // generate configuration
        config := kafka.ConfigMap{
                "bootstrap.servers": brokers,                              ❷
                "security.protocol": "sasl_ssl",                           ❸
                "sasl.mechanisms":   "PLAIN",
                "sasl.username":     args["user"],                         ❹
                "sasl.password":     args["password"],
        }
        return &config
}
```

❶ Build a broker list in comma-separated format from the arguments.

❷ Use the broker list.

❸ Select the security protocol.

❹ Pass username and password.

This configuration is specific to a cluster in the IBM Cloud; however, since there are a wide variety of parameters, you will have to choose the best ones for your particular Kafka configuration. You should check the documentation about the possible configuration options on GitHub.

Once you have the configuration, you can build a producer. Since we can use a producer multiple times, we need a variable to cache it. We also need a channel to receive acknowledgment of message delivery:

```
var producer *kafka.Producer

var deliveryChan chan kafka.Event
```

We are now ready to write the producer that puts everything together:

```
// Producer returns a producer to Kafka in a persistent way
func Producer(args map[string]interface{}) *kafka.Producer {
        if producer != nil {
                return producer                                            ❶
        }
        // create a producer and return it
        p, err := kafka.NewProducer(configProducer(args))                  ❷
        if err != nil {
                log.Println(err)
                return nil
        }
        producer = p                                                       ❸
        deliveryChan = make(chan kafka.Event)                              ❹
```

```
        return producer
}
```

❶ Return a cached producer.

❷ Create a new producer.

❸ Cache the producer.

❹ Create the channel for sending events.

Sending a Kafka Message

Once you have a producer you can send messages. A message in Kafka is just a byte array. However, to send it, you also have to specify the topic (a string) and a partition (an integer). The partition can be a special value, such as `kafka.PartitionAny` or a specific value.

In our chat application we pick one (and only one) partition because we do not want our messages to be split. We need to ensure all the messages are delivered to each consumer. Kafka ensures the messages are delivered to the consumers assigned to one specific partition, so it is important to pick the partition correctly.

The `Send` function receives a producer, a topic, a partition, and the message. It builds a `kafka.Message` with the topic and the partition, then sends it with the `kafka.Producer`. It passes a channel to receive an acknowledgment that the message was delivered.

The delivery in general is asynchronous, so you send a message and forget about it. If you want a synchronous call, you can wait for an acknowledgment that a message was successfully sent on the delivery channel.

In our application, we want the messages to be sent in order. To avoid messages being sent out of order we make the send synchronous and wait for the acknowledgment of the message delivery. We assume that each client will wait for the acknowledgment before sending another message.

Here's the code for sending a message:

```
// Send a message
func Send(p *kafka.Producer, topic string, partition int, message []byte) error {
        tp := kafka.TopicPartition{                                              ❶
```

```
                Topic: &topic,
     Partition: int32(partition),
    }
           msg := &kafka.Message{                                    ❷
                   TopicPartition: tp,
                   Value:          message,
           }
    p.Produce(msg, deliveryChan)                                     ❸
           e := <-deliveryChan                                       ❹
           m := e.(*kafka.Message)
           return m.TopicPartition.Error                             ❺
    }
```

❶ Create the `struct` to hold the topic and partition.

❷ Create the actual message to deliver.

❸ Send the message using the channel.

❹ Wait for acknowledgment that the message has been sent.

❺ Return the error.

Writing a Sender Action

Now that we have the functions we need to create a producer and use it, wrapping the complexities of interacting with Kafka. We can write, deploy, and test a `sender` action exposed as a web action. Since we are going to use the action in a chat application, we protect it with a password. However, since we want to avoid the complexity of setting a separate store for the password, we will only check it against a fixed secret we store as a parameter of the action when deploying it.

Let's write the `sender` action. It starts with checks: first we check for the password, then we check the arguments. Then we create a producer, since we need it to deliver messages. Finally, once we have the channel, we can send the messages and return an "OK" response. But before we go into the details, let's write a simple helper function for returning errors, which makes the code simpler to write:

```
package main

func mkErr(message string) map[string]interface{} {
        return map[string]interface{}{
                "body": "ERROR: " + message,
        }
}
```

Now we can write the code for the `main` function of the action:

```
// Main is the entry point for the sender action
func Main(args map[string]interface{}) map[string]interface{} {

        pass, ok1 := args["pass"]                                          ❶
        secret, ok2 := args["secret"]
        if !(ok1 && ok2 && pass == secret) {
                return mkErr("authentication failed")
        }


        // get args
        message, ok := args["message"].(string)                            ❷
        if !ok {
                return mkErr("no message")
        }
        topic, ok := args["topic"].(string)
        if !ok {
                return mkErr("no topic")
        }
        partition, ok := args["partition"].(float64)
        if !ok {
                return mkErr("no partition")
        }

        // retrieving the connection
        p := Producer(args)                                                ❸
        if p == nil {
                return mkErr("cannot connect")
        }
        // sending the message
        err := Send(p, topic, int(partition), []byte(message))             ❹
        if err != nil {
                return mkErr(err.Error())
        }
        return map[string]interface{}{                                     ❺
                "body": "OK",
        }
}
```

❶ Check the password against a secret.

❷ Extract the mandatory `topic`, `partition`, and `message` parameters.

❸ Create the producer, which can be cached, using other parameters.

❹ Send the actual message.

❺ If you get here everything is OK.

Deploying and Testing the Producer

Our action is a Go action. The steps to compile and test it were described in the previous chapter, but let's review them again here:

1. Collect a third-party library with `dep` to generate a *vendor* folder.
2. Precompile the action for efficient deployment.
3. Deploy it using the *cred.json* to connect to che kafaa action.
4. Being a web action, test it with `curl`.

To retrieve the `confluent-kafka-go` library and set up the *vendor* folder, you need to start from the following layout. Note that you need the *src/main* for the Go conventions (a *vendor* folder cannot be within the top-level folder). You also need the `dep` tool to collect the dependencies:

```
cred.json
sender
└── src
    └── main
        ├── send.go
        └── sender.go
```

The commands to collect the dependencies and to compile and deploy the action are:

```
$ cd sender/src/main
$ GOPATH=$PWD/../.. dep init                                          ❶
Using ^0.11.6 as constraint for direct dep \
 github.com/confluentinc/confluent-kafka-go
Locking in v0.11.6 (460e8e4) for direct dep \
github.com/confluentinc/confluent-kafka-go
$ cd ..
$ zip - -qr main | docker run -i openwhisk/actionloop-golang-v1.11 \  ❷
  -compile main>../sender.zip
$ cd ..
$ wsk action create mesg/sender sender.zip \                          ❸
 -P ../cred.json -p topic queue-p partition 0 \
 -p secret s3cr3t \
 --kind go:1.11 --web true
ok: created action mesg/sender
$ SEND=$(wsk action get --url mesg/sender | tail -1)                  ❹
```

❶ Collect dependencies with the `dep` tool.

❷ Precompile the sources in a single binary.

❸ Deploy the action, specifying Kafka credentials, our topic and partition, and the secret.

❹ Save the URL to access the `sender` action.

You can now test if the action works correctly and send messages. You need to use two terminals to verify this:

- In terminal 1, execute `wsk activation poll mesg/logme`.
- In terminal 2, execute `curl "$SEND?message=hello&pass=s3cr3t"`.

If you see the expected output, your `sender` action is ready.

A Kafka Consumer in Go

Since you now know how to send messages to a topic and a partition, the next logical step is learning how to receive messages. Kafka stores a message in a partition of a topic and keeps it for a configurable amount of time. When you create topics, you also specify how long messages should be retained. When you want to read messages from Kafka you have to create a consumer that subscribes to a topic. Each consumer is placed in a group using an identifier called `group.id`. A group is then assigned to a partition, either asking explicitly for a given partition or letting Kafka select the partition.

If there are too many consumers assigned to one partition, Kafka can rebalance the consumers, reassigning them to other partitions.

Note that a consumer group tracks a partition of a topic, and when one consumer of a group reads a message, the other consumers in that consumer group will not see it anymore. In other words: *each consumer of a consumer group consumes one message of a partition of a topic*, hiding it from other consumers of the group. All messages of a topic are numbered, and Kafka remembers which message to read.

When you first create a consumer in a consumer group, Kafka automatically initializes the latest message so that only new messages are read. It is, however, possible to read messages starting from the earliest available or even from a specific timeframe.

Creating a Consumer

Now let's create a consumer. Since we want to use this function for a chat application, there are some constraints we need to keep in mind.

To avoid losing messages, we use only a single partition, so we specify the topic and the partition for each consumer in a specific consumer group. OpenWhisk executes

actions in multiple runtimes; it is the system that decides whether to reuse a runtime that is already in existence or create a new one, so we can have multiple runtimes to serve a single action.

To improve efficiency, when creating a consumer we will cache it, but we assign a consumer group name for each user so even if there are multiple runtimes for executing the action, messages are still seen in order and only once.

Using our Go library, we create the consumer in two steps: first we prepare a `ConfigMap` passing all the configuration parameters, then we create the consumer. For convenience, we split the operation into two functions: `configConsumer` to prepare the configuration and `Consumer` to create the consumer.

In the following, most of the code is similar to the `configProducer` function already described in "Creating a Producer" on page 301, so I do not comment on it again. Instead, I focus on describing the additional details required for the consumer:

```
func configConsumer(args map[string]interface{}) *kafka.ConfigMap {

        // extract broker list from map
        brokers := ""
        for _, s := range args["kafka_brokers_sasl"].([]interface{}) {
                brokers  = s.(string)   ","
        }
        brokers = brokers[0 : len(brokers)-1]

        // generate configuration
        config := kafka.ConfigMap{
                "bootstrap.servers":               brokers,
                "security.protocol":               "sasl_ssl",
                "sasl.mechanisms":                 "PLAIN",
                "sasl.username":                   args["user"],
                "sasl.password":                   args["password"],
                "auto.offset.reset":               "latest",          ❶
                "go.events.channel.enable":        true,              ❷
                "go.application.rebalance.enable": false,             ❸
                "enable.auto.commit":              true,              ❹
        }
        return &config
}
```

❶ Initialize the offset to the latest available message.

❷ Allow the consumer to read messages using Go channels.

❸ Disable rebalancing, to keep consumers on the assigned partition.

❹ Enable autocommit so the offset is advanced when you read a message.

The consumer takes a configuration, a topic, a partition, and a group. It first tries to find a cached consumer for the given group. If it doesn't find one, it builds one, caches it, and assigns it to a given topic/partition combination:

```
var consumers = map[string]*kafka.Consumer{}

func Consumer(config *kafka.ConfigMap, \
    topic string, partition int32, group string) \
    *kafka.Consumer {
        // return cached consumer, if any
        if consumer, ok := consumers[group]; ok {                    ❶
                return consumer
        }

        // not found in cache,
        // create a consumer and return it
        config.SetKey("group.id", group)                             ❷
        consumer, err := kafka.NewConsumer(config)
        if err != nil {
                log.Println(err)
                return nil
        }

        // assign to a specific topic and partition
        assignment := []kafka.TopicPartition{{                       ❸
                Topic:     &topic,
                Partition: partition}}
        consumer.Assign(assignment)

        // cache consumer and return it                              ❹
        consumers[group] = consumer
        return consumer
}
```

❶ Search for the consumer in the cache; it is a map indexed by the group name.

❷ Instantiate the consumer with the configuration, augmented with the group id.

❸ Create a pointer to a topic and partition, the offset defaults to the value chosen by the options specified in the `ConfigMap`, then assign the consumer.

❹ Cache and return the consumer.

Receiving a Message

Once you have a consumer, you can receive messages. Our library allows two ways to get messages: one is by invoking the blocking `Poll` call, and the other is by receiving events on a channel. The `Poll` call is a blocking one, but you can specify a timeout; if you do not receive messages within the timeout the call will return.

We'll use the more idiomatic Go implementation using channels. The Go way is to listen on a channel (with the `<-` syntax) returned by `Events` wrapping the call with a `select`. In this way, if there are no messages in the channel the `default` option is picked, implementing a nonblocking call. Furthermore, the event returned by the `Events` call can be one of several different types. In this context, we're only interested in a `kafka.Message`. In Go, you use a switch with the `.(type)` syntax to perform different operations according to the underlying type returned by a generic method.

Using these techniques, we can write a `Receive` function that collects all the pending messages and returns an array when there are no more messages. The code is admittedly using nontrivial Go programming techniques:

```
func Receive(c *kafka.Consumer) []string {
  messages := []string{}                                          ❶
  for {                                                           ❷
         select {                                                 ❸
          case ev := <-c.Events():                                ❹
                switch e := ev.(type) {                           ❺
                   case *kafka.Message:                           ❻
                     v := string(e.Value)                         ❼
                            messages = append(messages, v)        ❽
                }
          default:
                 return messages                                  ❾
         }
  }
}
```

❶ Prepare an empty slice to return.

❷ Loop forever; loop terminated by inner returns.

❸ This `select` allows us to be nonblocking on the channel read.

❹ Receive events from Kafka through a channel.

❺ This `switch` with `the type allows us to determine the type of message.

❻ Receive a message.

❼ Extract the actual value from the message as a string.

❽ Append to the slice we are going to return.

❾ Return the collected messages when there are no more messages to receive.

Writing a Receiver Action

The `Consumer` and `Receive` functions are fine from receiving messages from Kafka. But now we want to wrap them in a web action to use in a web chat application. Since we are going to expose this action to the public internet, we want to protect it with a password, in the same way as we did for the `sender` action. Again, to avoid the complications of setting up a database for storing passwords we'll use just one fixed password, stored in the parameter `secret` provided at deployment. The `topic` and `partition` parameters are also provided at deployment time.

To use the consumer, a client must provide the parameters `pass` and `group`, which identifies a consumer group name. In this way, a client using the same group name will see all the messages since the first use of the group name. The first use of a group is a sort of "log in" procedure. If the client does not provide a `group` parameter, then a new random group name is generated and returned. This name should be used for all the subsequent interactions with the same client.

Let's see the implementation. The action is a web action and we want to return a JSON response, so, it must return a map with a key `body` for the body of the answer, and another key `headers` specifying the content type. Since all the answers of our action will be JSON, it makes sense to have a utility function `mkBody` to build such a JSON answer:

```
func mkBody(key string, value interface{}) map[string]interface{} {
        return map[string]interface{}{
                "body": map[string]interface{}{
                        key: value,
                },
                "headers": map[string]interface{}{
                        "Content-Type": "application/json",
                },
        }
}
```

We'll also want a utility function to generate the random numbers we use as group names when we need to create these as part of our authentication scheme. So, let's prepare a generator of random numbers:

```
// generate a random number
var generator = rand.New(rand.NewSource(
     time.Now().UnixNano()))                                    ❶
```

❶ Use the current time as a seed for the random number generator.

Now you can see the implementation of the `main` action. As a reminder, its type is:

```
func Main(args map[string]interface{}) map[string]interface{}
```

The action is a bit long, so we'll split it into snippets and discuss them separately.

The action starts by checking the password against the secret (as we did for the `sender`), then it reads the mandatory parameters for `topic` and `partition`:

```
// check password
pass, ok1 := args["pass"]
secret, ok2 := args["secret"]
if !(ok1 && ok2 && pass == secret) {
    return mkBody("error", "authentication failed")
}

var topic string
var partition float64
var ok bool

// check if there are a topic and partition
topic, ok = args["topic"].(string)
if !ok {
    return mkBody("error", "topic required")
}
partition, ok = args["partition"].(float64)
if !ok {
    return mkBody("error", "partition required")
}
```

Now we can implement the core logic, as described before. If there is no parameter `group`, the action generates a random group name, then creates the consumer. The consumer is cached so we create it only once for the action runtime.

If instead the client invokes the receiver providing the group name, the consumer is retrieved from the cache and used to read the messages. The code uses the functions `Consumer` and `Receive` from before:

```
// check there is a group then handle receive
if group, ok := args["group"].(string); ok {                          ❶
        config := configConsumer(args)
        consumer := Consumer(config, topic, int32(partition), group)  ❷
        return mkBody("messages", Receive(consumer))                  ❸
} else {
        // no group, generate a group name
        group := fmt.Sprintf("g%d", generator.Uint64())               ❹
        return mkBody("group", group)                                 ❺
}
```

❶ There is a group name.

❷ Create or retrieve a cached consumer.

❸ Retrieve and send the messages.

❹ No group provided, so generate one.

❺ Return the body of the message.

When there are too many requests, OpenWhisk creates more instances of the action runtime to serve those clients. In this case, when retrieving the consumer, it will not be in the cache so the new instance creates another consumer. However, since it uses the same consumer group, it maintains continuity and order in the messages.

Usage of an action exposed to the public internet can grow rapidly. OpenWhisk will spawn more action runtimes to handle the load, in turn creating more connections to the Kafka brokers. At a certain point, the brokers may stop accepting new connections. The technique shown here is therefore not suitable for high scalability.

Testing the Consumer

You can now deploy and test the consumer in the same way as described in "Deploying and Testing the Producer" on page 306; you only have to change `sender` to `receiver` and deploy the action as `mesg/receiver` with the same parameters, including the secret. Let's now look at a simple test of the action, with the help of the `sender` action:

```
$ SEND=$(wsk action get --url mesg/sender | tail -1)                    ❶
$ RECV=$(wsk action get --url mesg/receiver | tail -1)
$ curl "$RECV"                                                          ❷
{
  "error": "authentication failed"
}
$ curl "$RECV?pass=s3cr3t"                                              ❸
{
  "group": "g3951381834954569784"
}
$ curl "$RECV?pass=s3cr3t&group=g3951381834954569784"                   ❹
{
  "messages": []
}
$ curl "$SEND?pass=s3cr3t&message=\[1\]hello"                           ❺
OK
$ curl "$SEND?pass=s3cr3t&message=\[2\]hi"                              ❻
OK
$ curl "$RECV?pass=s3cr3t&group=g3951381834954569784"                   ❼
{
  "messages": ["[1]hello", "[2]hi"]                                     ❽
}
$ curl "$RECV?pass=s3cr3t&group=g3951381834954569784"                   ❾
{
  "messages": []
}
```

❶ Get the URLs of the `sender` and `receiver` actions.

❷ Invoke the action without a password, resulting in an authentication error.

❸ Invoke the `receiver` action without the group, returning a new consumer group name.

❹ Try to use the new group name; it is empty.

❺ Send a message to the topic.

❻ Send another message.

❼ Try to retrieve the messages.

❽ Now both the messages are available in the topic `queue`.

❾ The topic is now empty.

Implementing the Web Chat Application

The two actions developed so far (`mesg/sender` and `mesg/receiver`) are the building blocks required to implement the simple web chat application shown in Figure 11-5.

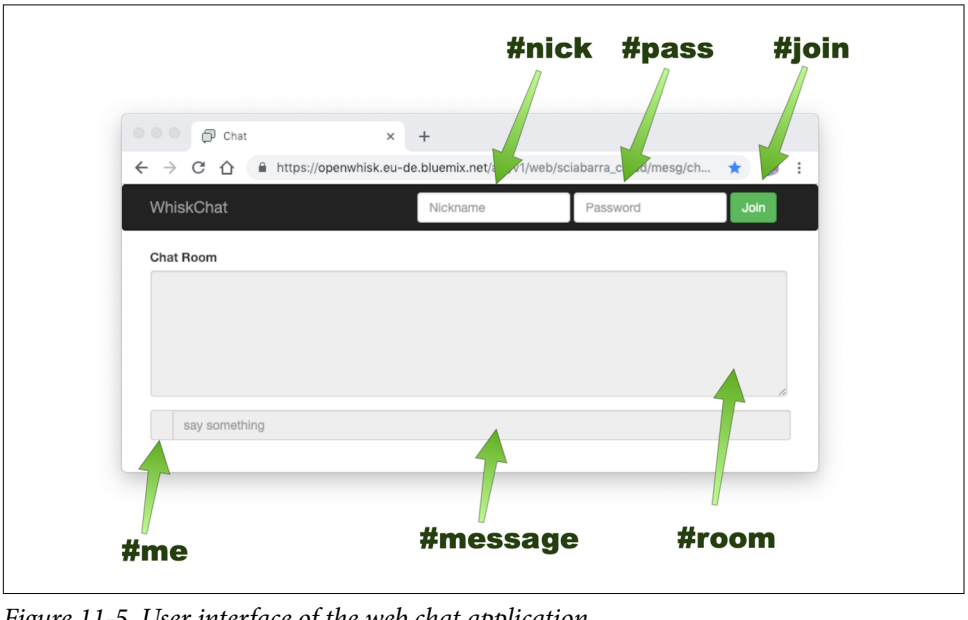


Figure 11-5. User interface of the web chat application

This chat is a web action; users of the chat access it with their browser and must join it, providing an arbitrary nickname and the password (which is the same for all users in our simple implementation). Once logged in, users can read messages from others in the main text area, and they can add their own.

Overview

We implement the chat app by leveraging the two actions developed before for sending and receiving messages in Kafka. The application is a SPA (single-page application), and we reuse the technique in "Embedding Resources" on page 274. The Go code for serving the embedded resources is the same as described there; the only differences are in the client part.

Here is the folder hierarchy:

```
chat
├── res
│   ├── bootstrap.min.css
│   ├── bootstrap.min.js
│   ├── favicon.ico
│   ├── index.html
│   ├── jquery-1.12.4.min.js
│   └── main.js
└── src
    ├── app
    │   ├── Gopkg.lock
    │   ├── Gopkg.toml
    │   ├── a_app-packr.go
    │   ├── assets.go
    │   ├── box.go
    │   └── vendor
    └── main.go
```

Here, the subfolder *src* uses the same code as before to embed static assets in a Go action. In the subfolder *res* we put the resources for implementing the client part of the chat app. In particular, you can see:

- The CSS library bootstrap
- The JavaScript library jQuery
- The HTML of our application, *index.html*
- The JavaScript of our application, *main.js*

We do not cover the libraries in detail, since both are widely known, but instead focus on describing the HTML user interface of the application and then the JavaScript logic.

User Interface

Figure 11-5 shows the main parts of the user interface as well as the HTML id of each component. The user interface has two parts: the heading, used only for logging in, and the main body. A simplified version (I've removed style details for easier reading) of the HTML of the heading follows:

```
<form id="form">                                                    ❶
  <input id="nick" type="text"                                      ❷
    placeholder="Nickname">
  <input id="pass" type="password"                                  ❸
    placeholder="Password">
  <button id="join"                                                 ❹
   type="button">Join</button>
</form>
<label for="comment">Chat Room</label>
<textarea class="form-control"                                      ❺
  rows="15" id="room" readonly></textarea>
<span id="me" class="input-group-addon"></span>                     ❻
<input id="message" type="text"
  placeholder="say something" disabled="true">                      ❼
```

❶ Container for the user interface section to join the chat.

❷ Nickname input text box.

❸ Password input text box.

❹ Join button.

❺ Text area to show received messages.

❻ Area displaying the current nickname.

❼ Input field to send messages.

Note that the message and room input fields are initially disabled.

Initializing

The HTML of the web chat application also includes the *main.js* file, discussed in this section (in separate snippets as before). The chat application uses the actions `sender` and `receiver` to perform its work. The action `chat` is just the container of the client side and acts as a static web server (deployed as an action for convenience).

The client code needs to locate the URLs of the other two services. You can install the three actions in any OpenWhisk server; however, I assume that `chat`, `sender`, and `receiver` are all deployed in the same server, namespace, and package so that you can

find the URLs of the other two from the URL of one of them. The JavaScript code uses this assumption to locate the URLs of the other services. If the URL of the HTML page is, for example

```
https://openwhisk.example.com/api/v1/web/my_namespace/mesg/chat/index.html
```

you can locate the `sender` and the `receiver` by reading the whole URL, removing `/chat/index.html` and adding `/sender` or `/receiver`.

Translated into JavaScript, this is:

```
var base = location.href
base = base.substring(0, base.lastIndexOf("/"))
base = base.substring(0, base.lastIndexOf("/"))
var sender = base + "/sender"
var receiver = base + "/receiver"
```

Once we have initialized the variables for locating the services, we also initialize the global variables to contain the nickname, password, and group. Those variables will get proper values when a user joins the chat:

```
var nickname = ""
var password = ""
var group = ""
```

The application uses jQuery, so we complete the initialization by registering two handlers: one to handle the click the Join button and another to send messages to the chat, intercepting keystrokes in the message input field. Let's do just this for now (we'll see the functions `join` and `message` later):

```
$(function() {
        $("#join").click(join)
        $("#message").keyup(message)
})
```

Note that initially the input field to send chat messages is disabled. The user can only click the Join button.

Joining

The user fills in the two fields, "nickname" and "password," and then clicks the Join button. The function `join` is an event handler registered to handle the click of the button. It reads the fields and checks to see if they are empty. If the user provided values and they are not empty, the URL of the `mesg/sender` action is invoked, passing the password. The client code never sends the nickname to the backend; it is used only on the client side and stored in the local variable `nickname`.

Here is the code for the `join` function:

```
// join the chat
function join() {
```

```
        nickname = $("#nick").val()                                    ❶
        password = $("#pass").val()
        if(nickname == "" || password == "") {                         ❷
                alert("Please specify nickname and password")
                return
        }
        // first connection, get the group
        $.post(receiver,                                               ❸
                {"pass": password},
                joined)
}
```

❶ Read the fields with the nickname and password.

❷ Validate the values.

❸ Invoke the backend action.

When the first invocation of the `sender` action completes, if it is successful (the password was correct) it returns the group name and stores it in a local variable. At this point, you need to get ready for the user to start chatting with others. You also need to hide the login form to avoid multiple logins and enable the text area to write the messages. But the most important step is starting to receive messages. Since currently OpenWhisk does not support any form of invocations pushed by the server (like WebSockets), you have to use a pull mechanism. So you enable a repeated invocation of the `poll` function (discussed next) to read the upcoming messages:

```
// chat joined
function joined(data) {
        if(data.group) {
                group = data.group                                     ❶
                $("#form").hide()                                      ❷
                $("#me").text(nickname)                                ❸
                $("#message").removeAttr("disabled")                   ❹
                $.post(sender,                                         ❺
                        {"message": "**** "+ nickname + " joined ****",
                         "pass": password})
                setInterval(poll, 2000)                                ❻
        } else if(data.error)                                          ❼
                alert(data.error)
}
```

❶ Save the group name in a global variable.

❷ Hide the entire login form.

❸ Display the user's nickname before the input text field.

❹ Enable the input text field.

❺ Notify everyone in the chat of the new user.

❻ Enable the invocation of the `poll` function every 2 seconds.

❼ Display error messages, if any.

Receiving

After the user joined, the `poll` function executes every two seconds with the task of updating the central area of the chat with the messages received from other members. What it does is invoke the `receiver` action, using the group name and the password. The `receiver` returns all the messages it can read in the topic. Since there is a unique consumer group name for each member of the chat, the member is guaranteed to receive messages starting from the time the user logged in:

```
function poll() {
  $.post(receiver,                                                    ❶
         {"group": group, "pass": password},
    function (data) {
                if(data.error) {                                      ❷
                        console.log(data.error)
                        return
                }
                if(data.messages.lenght==0)                           ❸
                        return
                var curr = $("#room").val()                           ❹
                for(message of data.messages) {
                        curr += message + "\n"
                }
                $room = $("#room")
                $room.val(curr)
                $room.scrollTop($room[0].scrollHeight)                ❺
        })
}
```

❶ Make an Ajax request to read messages, passing the group and password.

❷ Something went wrong—log and continue.

❸ There are no messages; just return.

❹ There are messages, so read the current text area, append the messages, and update.

❺ Ensure the text area scrolls to the end so the user can see the messages added.

Sending

Finally, let's examine the handler for sending messages. In the initialization step, we attached an event handler to the message text input to inspect keystrokes. The purpose of this handler is to wait for the user to press Enter. When this happens, the event handler does an Ajax call to the `sender` action. It passes the password and constructs a message prefixing the nickname (that the code manages entirely client-side). It waits for the action to complete and only checks if the returned value is `OK` (otherwise it displays an error message):

```
function message(e) {
        message = $("#message").val()                                    ❶
        if(message == "")
                return
        if(e.keyCode === 13) {                                           ❷
                $("#message").val("")                                    ❸
                $.post(sender,                                           ❹
                        {"message": "["+nickname+"] "+message,
                         "pass": password },
                        function (data) {                                ❺
                          if(data!="OK")
                                alert(data)
                        }
                )
        }
}
```

❶ Check that there is a message, actually.

❷ Check whether the user pressed Enter.

❸ Clear the message input text.

❹ Ajax invocation of the `sender` action, with a proper message and the password.

❺ Show an error message if the value returned was not `OK`.

Summary

In this chapter we explored how to use Kafka as a messaging tool for OpenWhisk and covered examples using Go. We started looking at Kafka and its main concepts, like topics, partitions, and consumer groups, and we tested it interactively using the `messaging` package and the `kafkacat` tool. Then we went into detail about how to write Go actions able to read and write in Kafka. Finally, we wrote a simple web chat application that leveraged Kafka features to keep track of users and distribute messages to members of the chat.

CHAPTER 12

Deploying OpenWhisk with Kubernetes

Apache OpenWhisk is available out of the box in the IBM Cloud and in the Adobe Cloud as I/O Runtime, but it is a true open source project and can actually be deployed everywhere. It can be installed in other clouds, either public or private, or on your own servers. In this chapter, you will learn how to install OpenWhisk both for development and production, and how to deploy complex applications in it. But before going into all the details, let's cover some of the basics.

OpenWhisk is actually built on Docker. Docker is known as a "lightweight" virtualization environment. It provides a format for defining disk images, named Docker images, and a way to launch them.

There is a public distribution point known as Docker Hub. OpenWhisk is distributed on Docker Hub in the form of a collection of Docker images, deployed to create a "control plan." When you deploy an action, you communicate with the OpenWhisk control plan, and it creates other containers to run actions. Runtimes to execute actions are also Docker images.

Generally, Docker alone is not enough to run complex, multiserver applications. You need another piece of software, called *Kubernetes*. In this chapter, we focus on running OpenWhisk in Kubernetes, either locally for development purposes, in the cloud, or on a bare metal server.

Docker is essentially a Linux product, but a version for Windows and Mac (called Docker Desktop) exists. Docker Desktop also includes Kubernetes. Linux versions are generally used for production and building clusters, whereas the Windows/Mac versions are used for development, running a single-node Kubernetes cluster.

In this chapter, we first install Kubernetes, and then install OpenWhisk in Kubernetes.

The source code for the examples related to this chapter is available on GitHub.

Installing Kubernetes

Figure 12-1 shows the architecture of a Kubernetes cluster and depicts what we will deploy in this chapter.

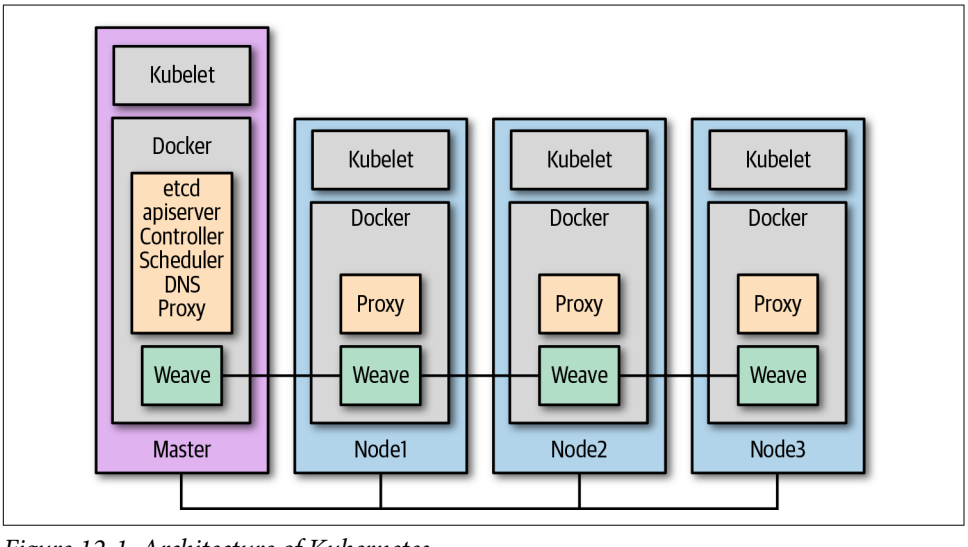


Figure 12-1. Architecture of Kubernetes

As you can see, a cluster is comprised of a number of "nodes" (which are separate servers, either VMs or physical servers) all running Docker. In addition, each node also runs a Kubernetes-specific service called a *kubelet*. Everything else runs inside Docker as a Docker container.

One node is the *master* node, acting as the coordinator, while the others are *worker* nodes. The master node generally does not execute any useful work but just supervises the worker nodes. Also note that in each worker node, there are two Kubernetes-specific containers: one for the networking connection (here, we use `weave`) and another one, called a *proxy*, that allows each container to provide external services in a node-independent manner.

Installation Types

In this chapter, I describe multiple procedures to install Kubernetes and some related tools needed for OpenWhisk. In particular, we cover the installation of:

- A single-node Kubernetes cluster and some Kubernetes tools on your local machine, for Mac and Windows
- A multinode Kubernetes cluster on a cloud provider with VMs
- Kubernetes on a bare metal Linux server

Note that major cloud providers like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud offer Kubernetes as part of their packages. However, it is also possible to install Kubernetes on those cloud providers that do not provide it out of the box as is the case in this chapter.

Installing kubectl and Helm

Before going into the details of the various installation procedures, first let's download a couple of command-line Kubernetes tools: `kubectl` and Helm. You'll need these tools to control either your local Kubernetes cluster or a remote one.

`kubectl` does what the name suggests—it controls Kubernetes. In particular, it offers many commands to read, create, update, and delete resources in Kubernetes. Those resources are generally represented by a collection of descriptions in YAML format.

Although `kubectl` manages resources, it does not provide any features to organize and parametrize those resources. Since Kubernetes clusters are not all created equal, you'll always need to change some parameters in the configuration for each specific case. This is where Helm comes in. It is a tool used to manage Kubernetes resources as groups, specifying parameters to adapt to different needs.

You should install the latest available versions of these tools. To see the latest available version of `kubectl`, type:

```
$ curl https://storage.googleapis.com/\
kubernetes-release/release/stable.txt
v1.13.1                                                    ❶
```

❶ The actual version you see may be different.

For details on the latest version of Helm, check out its GitHub repository.

Let's assume you want to download binaries of `kubectl` version 1.13.1 and Helm version 2.12.1.

Both `kubectl` and `helm` are developed in the Go programming language and are available as binaries for all of the major operating systems. Generally, the name of the kernel of the operating system identifies the binary, then the architecture of the processor. For Linux and Windows, the name of the kernel is just `linux` or `windows`; for macOS it is `darwin` (as this is the actual name of the underlying operating system). For the architecture, currently, the more widely used is the X8_64 (the 64-bit version of the X86 instruction set) normally identified as `amd64`. It is also possible (but less common) to use a 32-bit version.

On macOS use the following commands to download and install the tools:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/\
  release/v1.13.1/bin/darwin/amd64/kubectl
$ chmod +x kubectl
$ sudo mv kubectl /usr/local/bin
$ curl https://storage.googleapis.com/\
  kubernetes-helm/helm-v2.12.1-darwin-amd64.tar.gz
$ tar xzvf helm.tar.gz --strip-component=1
$ sudo mv helm /usr/local/bin
```

For Linux, you should use the same commands, substituting `linux` for `darwin`. If you are using Windows, open the PowerShell as an administrator and type:

```
PS> $client = new-object System.Net.WebClient
PS> $client.DownloadFile("https://storage.googleapis.com/kubernetes-release/\
    release/v1.10.3/bin/windows/amd64/kubectl.exe",\
    "C:\Windows\kubectl.exe")
PS> $client.DownloadFile("https://storage.googleapis.com/kubernetes-helm/\
    helm-v2.12.0-windows-amd64.zip", "C:\helm.zip")
PS> Expand-Archive C:\helm.zip -DestinationPath C:\Windows\
PS> move C:\helm\windows-amd64\helm.exe C:\Windows\
```

To make sure the commands are properly installed and available on the path, invoke them to display their versions (these commands should work on all operating systems):

```
$ kubectl version --client --short
Client Version: v1.13.1
$ helm version -c --short
Client: v2.12.1+g02a47c7
```

Since you have not yet installed Kubernetes, you are asking only for the client version. You would get an error if you tried to connect to the server (Kubernetes).

Installing Kubernetes Locally

If you are using a Windows or Mac system as a development environment, you can easily install a single-node Kubernetes cluster using Docker Desktop, available for both platforms.

If you are using Linux you can follow the procedure described in "Installing Kubernetes on a Bare Metal Server" on page 339 to install Kubernetes on a server with just one master. There are other options, but they are not, in my opinion, any simpler than installing a virtual machine with Kubernetes as described here.

To download Docker Desktop, go to its download site. Register, follow instructions, and download Docker Desktop. On a Mac, once you hve Docker Desktop up and running, enable Kubernetes as depicted in Figure 12-2.

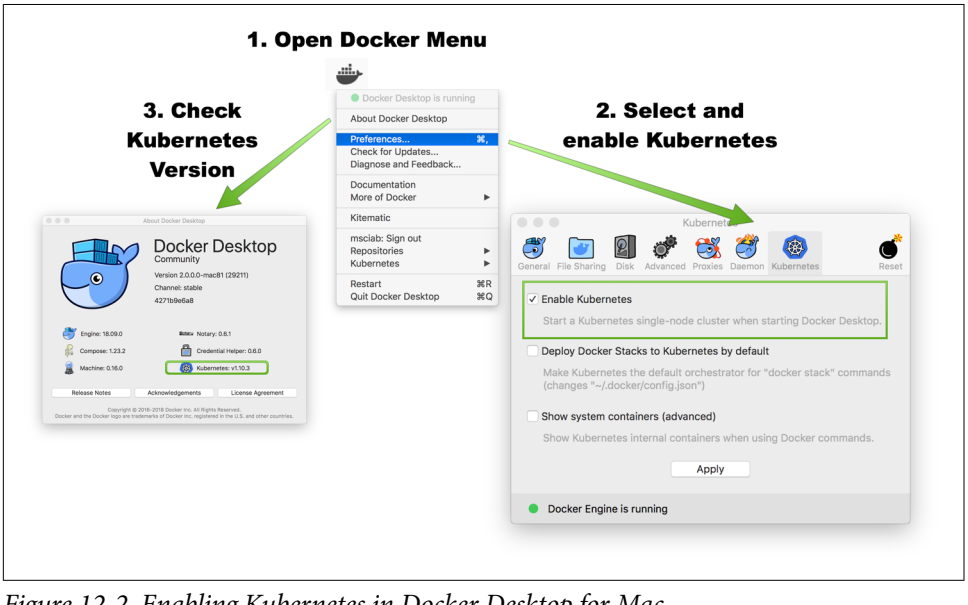


Figure 12-2. Enabling Kubernetes in Docker Desktop for Mac

The steps are as follows:

1. Click on the Docker icon in the menu bar and select Preferences.
2. Select the Kubernetes icon.
3. Check the Enable Kubernetes box and click Apply.
4. Now go back into the Docker menu and select About Docker Desktop.

5. Take note of the version of Kubernetes installed (in this case it is v1.10.3).

On Windows, after installing Docker Desktop you can enable Kubernetes as depicted in Figure 12-3.

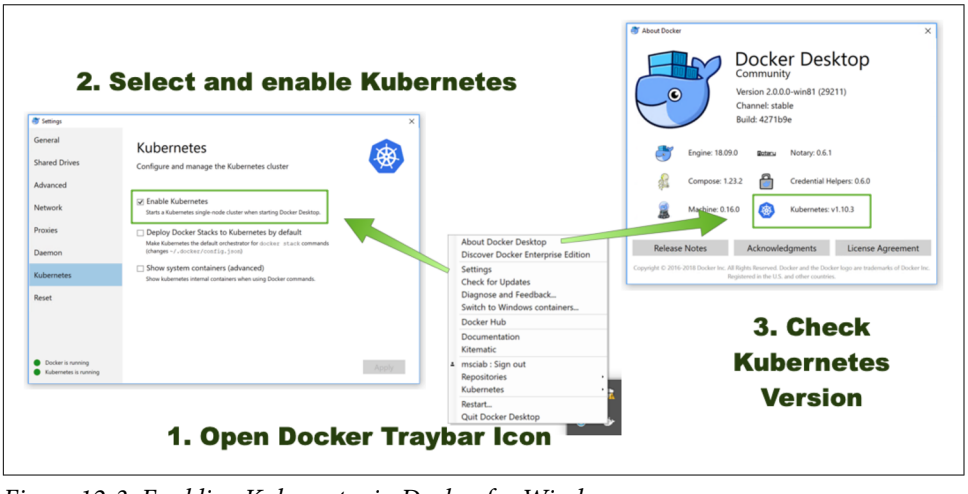


Figure 12-3. Enabling Kubernetes in Docker for Windows

Here are the steps:

1. Click on the Docker icon in the Windows taskbar and select Settings.
2. On the menu on the left, select Kubernetes.
3. Check the Enable Kubernetes box, and click Apply.
4. Now go back into the Docker menu and select About Docker Desktop.
5. Take note of the version of Kubernetes installed (in this case it is v1.10.3).

You can now check if you can connect to the local Kubernetes server. Type the following commands (either in Terminal or PowerShell) and check the results (your version numbers may be different):

```
$ kubectl version --short
Client Version: v1.10.3
Server Version: v1.10.3
$ kubectl get nodes
NAME                 STATUS    ROLES     AGE       VERSION
docker-for-desktop   Ready     master    5d        v1.10.3
```

You may have `kubectl` configured to access other Kubernetes clusters than Docker Desktop. If you do not see `docker-for-desktop` as the only node in the cluster, you need to change context with the command `kubectl config set-context docker-for-desktop`.

Now that you have Kubernetes up and running, if you are not interested in deploying a Kubernetes cluster in the cloud or on a local server, you can proceed to "Installing OpenWhisk" on page 345.

Installing Kubernetes in the Cloud

Cloud providers usually offer, among other products, either VMs or physical machines. In both cases, you have to deploy an operating system in it. Here we'll focus on installation in the cloud using VMs with Linux-based operating systems..

While you can install operating systems into VMs from scratch, it is generally an awkward task, so vendors make them available as so-called *cloud images* for VM use. A cloud image is a special version of an operating system designed to be easily deployed in a cloud environment.

The main difference from a "classic" operating system installation lies in the way you specify the initial configuration of the operating system itself. When you install from scratch on a physical server with a monitor and keyboard, you generally have an installer with a user interaface that guides you through the various options.

In the cloud, you do not have the luxury of an installer, so you have to resort to a scripted installation. The package most frequently used for this purpose on Linux-based operating systems is `cloud-init`. You can use it to launch a cloud image and feed to it some "user data" that the `cloud-init` package uses to initialize the system.

For our purposes, we'll use a configuration file that simplifies the task of creating a Kubernetes cluster ready for installing OpenWhisk. The procedure requires a cloud provider that specifically offers Ubuntu cloud images and supports initializing them with a `cloud-init` user-data script.

The procedure can be adapted to other Linux distributions, like those based on Red Hat, but here I'll provide step-by-step instructions for the Hetzner and AWS clouds.

Architecture of a Kubernetes Cloud Deployment

Before starting, you need to understand the network architecture of a cloud deployment. Cloud architectures can be very complex, so we focus only on the basics here, including:

- Private versus public networks

- Ports to use
- Public IP addresses and DNS names
- Sizing of VMs

Some cloud providers offer only servers accessible with public IPs. Others also offer the ability to keep your servers in a private network with private IPs. You can then use a public IP that will be forwarded to a private IP only for those servers you want to expose to the internet. This scenario is illustrated in Figure 12-4.

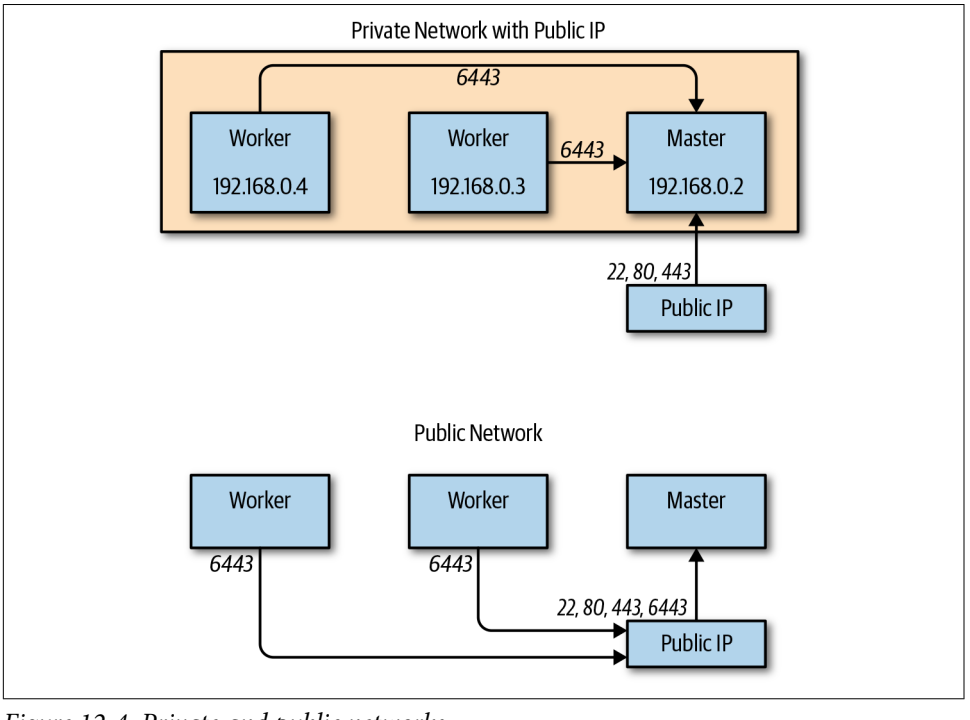


Figure 12-4. Private and public networks

Of course, a cluster is useful only if it can expose services to the world. In our case, OpenWhisk is entirely a web-based environment, so it will have to expose HTTP and HTTPS ports. In our deployment we'll add to the kit a frontend based on the reverse proxy Traefik, deployed on the same node as the master.

In order to be able to use the frontend, you need to open two ports: the standard port for HTTP, 80, and the standard port for HTTPS, 443. Since the installer is going to request a SSL certificate for HTTPS, you will also need a DNS name for the master. Generally, a DNS name is generated for each public IP, but you can always register the IP with your domain name if you prefer. This is important, since your applications will be exposed using that domain name.

Last but not least are the required resources. To install Kubernetes with a reasonable amount of resources, you need at least three virtual machines: one for the master with at least 4 GB of memory and two virtual CPUs (vCPUs), and two workers with at least 2 GB of memory each. I recommend, however, at least three workers with 4 GB of memory each.

Generic Procedure for Installing Kubernetes with cloud-init

Now let's take a look at the Kubernetes installation procedure. I'll explain it in general first, without focusing on a specific cloud provider. This approach can make the discussion a bit abstract and vague, though, so in the following sections I provide two practical examples using actual clouds, giving additional details. Before starting, you should ensure that you understand the architecture and the network requirements, as previously described.

Here are the steps for installing Kubernetes. First, create one server for the master with an image based on Ubuntu; I tested on Ubuntu 16.04 LTS and 18.04 LTS. You need at least 4 GB of memory and two vCPUs. The server must be able to communicate with the internet since it will download Kubernetes software from Google servers. Also, it must be accessible via SSH since you will have to log in to complete the installation.

Your cloud provider should provide a way to feed user-data for initializing your image. Ubuntu supports the `cloud-init` format for initialization, and an initialization script in this format that you can use for this purpose (e.g., by copying and pasting) is available in the GitHub repository.

If your provider does not assign a password to the instance (most providers do), you can set the root password on your own by uncommenting and editing the following lines at the beginning of the script:

```
chpasswd:
   list:
      - root:Change-me!                                    ❶
```

❶ Of course, change the password `Change-me!` to your password.

Since your server will be accessible from the public internet, it is highly recommended that you avoid easy-to-guess passwords like `root` or `password`.

Boot your server and wait until the boot process completes. Connect with `ssh` to the address provided by your provider. Once logged in, type:

```
cloud-init status --wait
```

At this point, you need to wait a bit (a minimum of a few minutes, but it can take up to 10). The system is updating the whole set of packages to the latest release and installing all the software required to run Kubernetes.

This command will show a sequence of dots, then once completed will disconnect your SSH connection. Reconnect (you may need to retry until the connection is available again) and type:

```
sudo kube-init <external-ip>
```

Here, the external IP is either the public IP of the VM or the public IP you have configured to forward traffic to the private IP (see "Architecture of a Kubernetes Cloud Deployment" on page 327 for details). In general, it should be the address you used to connect with `ssh`.

This installation script will run for a few minutes. If everything goes okay, it will produce output similar to the following:

```
*** If you have a private network, add to Cloud-Init for Workers:
runcmd:
-   kubeadm join 116.203.71.223:6443 --token mpkkl6.jbulhikuul4usw1v \
--discovery-token-ca-cert-hash sha256:7aff...
*** If you only have a public network, add to Cloud-Init for Workers:
runcmd:
- kubeadm join 116.203.71.223:6443 --token mpkkl6.jbulhikuul4usw1v \
--discovery-token-ca-cert-hash sha256:7aff...
```

When the system has only a public IP the two messages (for private and public networks) are identical. In general, however, the script emits both the command generated internally and the command "modified" for external use.

Now you have to pick the right initialization command for the other nodes. If you have a private network, you should use the first command, as the workers will connect to the master using the internal IP.

If instead you have a public network and both you and the workers can connect to the master only using the public IP address, you should use the second command.

In a private setup, workers connect to the master using a *different* IP than the one used to connect to the server, and you should not use the public IP as the workers may be unable to access it from the internal network.

Now you can create the workers. Your cloud provider should allow you to create multiple VMs with the same setup, so that you can create all the workers in a single step.

Create some servers (at least two) with at least 2 GB of memory. Copy the same *user-data.txt* file you used for the master, then edit it as follows. Insert the two lines shown in the output of the `kube-init` command at the beginning of the script, but *after* the `#cloud-config` line. For example:

```
#cloud-config                                                    ❶
runcmd:
- kubeadm join Y.Y.Y.Y:6443 --token aaa.bbb \                    ❷
  --discovery-token-ca-cert-hash sha256:ccccc
... (rest of user-data.txt) ...
```

❶ If this line is missing, `cloud-init` won't execute.

❷ Note that this line starts with `-`.

While the workers are starting and initializing you can monitor their creation from the master by executing this command:

```
sudo watch kubectl get nodes
```

You should see the nodes joining the cluster and reaching the ready state. Other servers will also appear in the list (again, all of this takes a few minutes to complete):

```
NAME    STATUS    ROLES     AGE     VERSION
kube0   Ready     master    20m     v1.11.6
kube1   Ready     <none>    5m55s   v1.11.6
kube2   Ready     <none>    6m49s   v1.11.6
```

If the whole process is not complete after 10 minutes, you can assume something went wrong. To troubleshoot, try to log in to the workers and see if you can reach the IP and the port specified in the `kubeadm join` command, as this is generally the problem.

Provisioning

Once you reach the ready, you have a working cluster—but you still need to install some components in the Kubernetes cluster before you can actually install OpenWhisk. In particular, you need a storage driver (for persisting data on disk) and an ingress controller (for accessing OpenWhisk via HTTPS).

IP and DNS name. You need also to assign an IP and a DNS name to your master server and request an SSL certificate. The DNS name is a prerequisite for the SSL certificate, and the `wsk` command in OpenWhisk is accessible over SSL.

The IPs assigned to VMs by cloud providers are generally floating. This means that is for some reason the server goes down, then your IP is lost. However, you can also generally get a stable IP (for a fee) that persists and can be moved from one server to another.

When you launch a virtual machine, cloud providers also normally provision ugly DNS names like `ec2-34-231-88-109.compute-1.amazonaws.com`. If the URL of your OpenWhisk server is not user-visible, you can use this, otherwise, you may want to register your own nicer-looking DNS name.

It is recommended at this point that you get a stable IP, assign it to the master server, and assign a nice-looking DNS name to this IP. You can find details on how to do this for Hetzner and AWS Cloud in the respective sections later in this chapter.

Rook and Traefik. Once you have configured your IP and DNS, you can execute on the server the following provisioning command:

```
$ kube-provision <public-dns-name> <your-email-address>
```

This command downloads a storage driver (Rook) and an ingress controller (Traefik). It also configures the ingress to get an SSL certificate using the free service Let's Encrypt.

Now, wait a bit and make sure the storage driver is installed before continuing. You can check the status by running this command:

```
$ sudo watch kubectl --namespace rook-ceph get pod
```

You should see something like this:

```
NAME                                READY   STATUS      RESTARTS   AGE
rook-ceph-mgr-a-579779457d-pr287    1/1     Running     0          19h
rook-ceph-mon-a-6bd4886f59-62n2k    1/1     Running     0          19h
rook-ceph-mon-b-78bb4d69f6-btdx5    1/1     Running     0          19h
rook-ceph-mon-c-b7cbd46b4-rlf8q     1/1     Running     0          19h
rook-ceph-osd-0-8446f8bc49-rn8pc    1/1     Running     0          19h
rook-ceph-osd-1-87856b466-24fc2     1/1     Running     0          19h
rook-ceph-osd-2-58544b9c89-8nj6t    1/1     Running     0          19h
rook-ceph-osd-prepare-kube1-cmkcz   0/2     Completed   0          19h
rook-ceph-osd-prepare-kube2-rgfvc   0/2     Completed   1          19h
rook-ceph-osd-prepare-kube3-g7ldj   0/2     Completed   1          19h
```

Make sure you can see at least as many `rook-ceph-osd-prepare` pods in the "completed" state as the worker nodes you have.

When your deployment completes, test it by creating a sample deployment with:

```
$ sudo kubectl apply -f /usr/local/etc/sample.yaml
namespace/sample created
persistentvolumeclaim/rook-claim created
pod/nginx-pod created
service/nginx-svc created
ingress.extensions/nginx-ingress created
```

You may need to be a bit patient—getting the SSL certificate may take a few minutes—but in the end at *https://<public-dns-name>/welcome* you should see a `Welcome!` message without getting SSL errors in your browser.

Congratulations! You have a Kubernetes cluster ready for installing OpenWhisk. Before we get to that, though, let's look at a few examples of how to use the generic procedure outlined here with some specific cloud providers.

Installing on Hetzner Cloud

The first example we'll look at uses the Hetzner Cloud. This cloud offers VMs based on cloud images including Ubuntu and supports initialization with `cloud-init` user data, so it is a good fit for our setup procedure.

Before starting, you have to configure an SSH key to access the servers. If you do not already have an SSH key, generate one with the command `ssh-keygen`. This command will ask you a few questions; if you accept the defaults it creates a private key in *$HOME/.ssh/id_rsa* and a public key in *$HOME/.ssh/id_rsa.pub*.

Now install the key as follows (see Figure 12-5):

1. From the Cloud Console, select the key icon in the menu on the left.
2. Click Add SSH Key.
3. Copy and paste the content of the file *$HOME/.ssh/id_rsa.pub*.

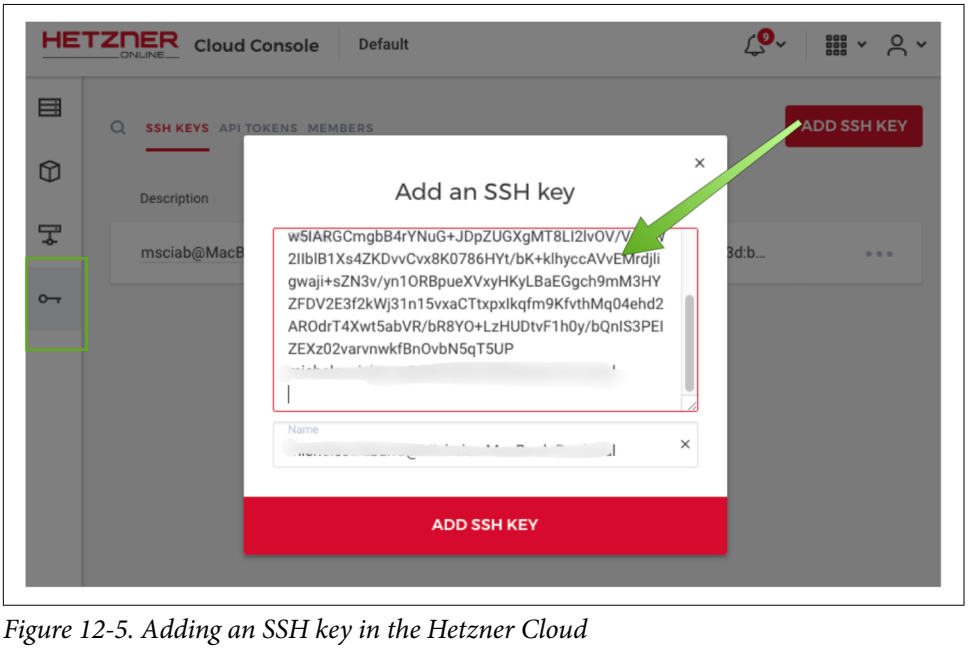


Figure 12-5. Adding an SSH key in the Hetzner Cloud

Once your public key is installed you can log in to the VMs and create the master as shown in Figure 12-6.

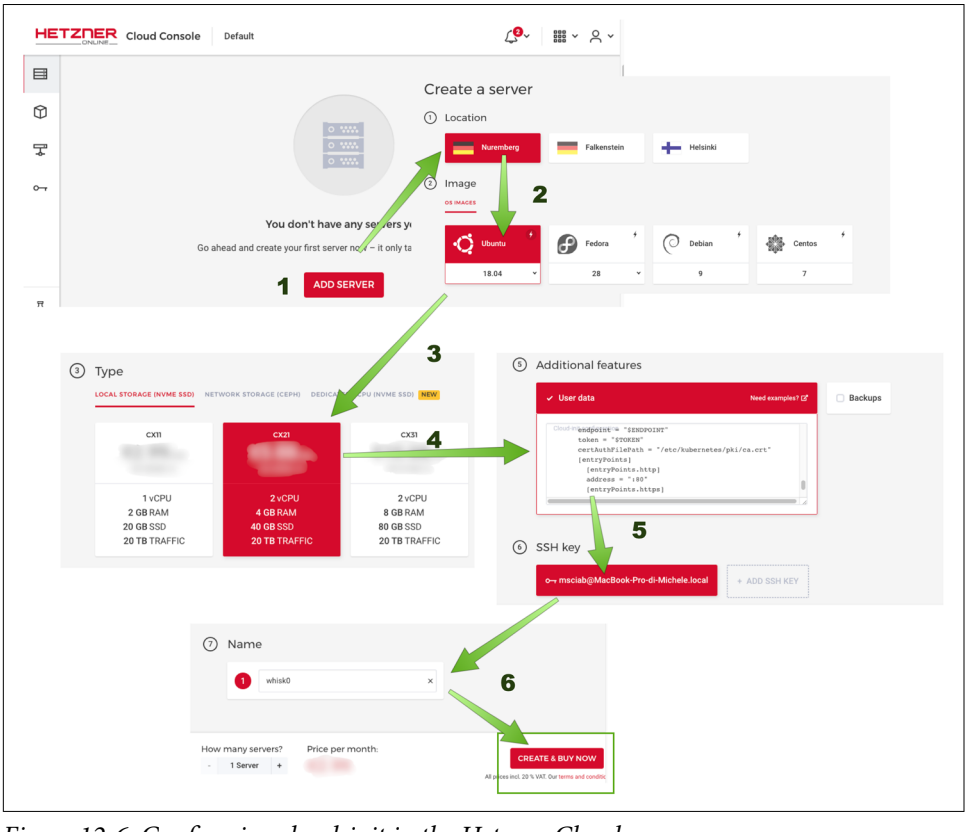


Figure 12-6. Configuring cloud-init in the Hetzner Cloud

Here are the steps:

1. In the Cloud Console, select Add Server.
2. Pick your location, then select Ubuntu.
3. Select the type. For the master, you need at least 4 GB RAM with two vCPUs.
4. Select "User data" and copy and paste the `cloud-init` initialization script into the text area.
5. Don't forget to select your SSH key; otherwise you will not be able to log in.
6. Name the server and launch it.

After you launch the server, it will show its public IP address. Connect to the server with `ssh root@<public-ip>`. If you created the private key with the `ssh-keygen`

command and it's stored in *$HOME/.ssh/id_rsa*, you should be able to log in without a password as the root user. Type `cloud-init status --wait`, and wait until the command terminates. Generally, at the end your SSH connection will be disconnected.

You can now reconnect and type the command `kube-init` *`<public-ip>`* to initialize the Kubernetes master. Once the master is ready, you use the same procedure to create the workers, one difference: in the user data text area, after pasting in the `cloud-init` script, you need to add after the `#cloud-config` line the two lines shown on completion of the `kube-init` script (be sure to use the second command, for a public network).

Now you can proceed with the rest of the installation procedure:

1. Return to the master and type `watch kubectl get nodes`.
2. Wait until all the nodes are in the ready state.
3. Complete the installation with the command `kube-provision` *`<dns-name>`* *`<your-email-address>`*.

In the Hetzner Cloud, each server gets a temporary IP and you will lose it if you have to recreate the server. So, it is advisable to allocate a so-called "floating" IP and assign it to the server. Furthermore, the Hetzner Cloud provides a DNS name for each server that you can use as the *`<dns-name>`* for provisioning your server. This will also be lost when you rebuild your VM. It is hence recommended that you get a DNS name (from any registrar) and use it to refer to your allocated IP.

Installing on AWS Cloud

Now let's see how to install Kubernetes on the AWS Cloud.

AWS includes a service called Elastic Container Service for Kubernetes (EKS) offering a ready-to-go Kubernetes cluster; there are also tools like `kops` that can be used to create a Kubernetes cluster on AWS. Both are alternatives to this procedure.

The required resources in AWS are not free. You will be charged for creating a cluster.

Before starting the installation process, install an SSH key on the AWS Cloud and locate the subnet id where you are going to install your VMs. Figure 12-7 shows these steps.

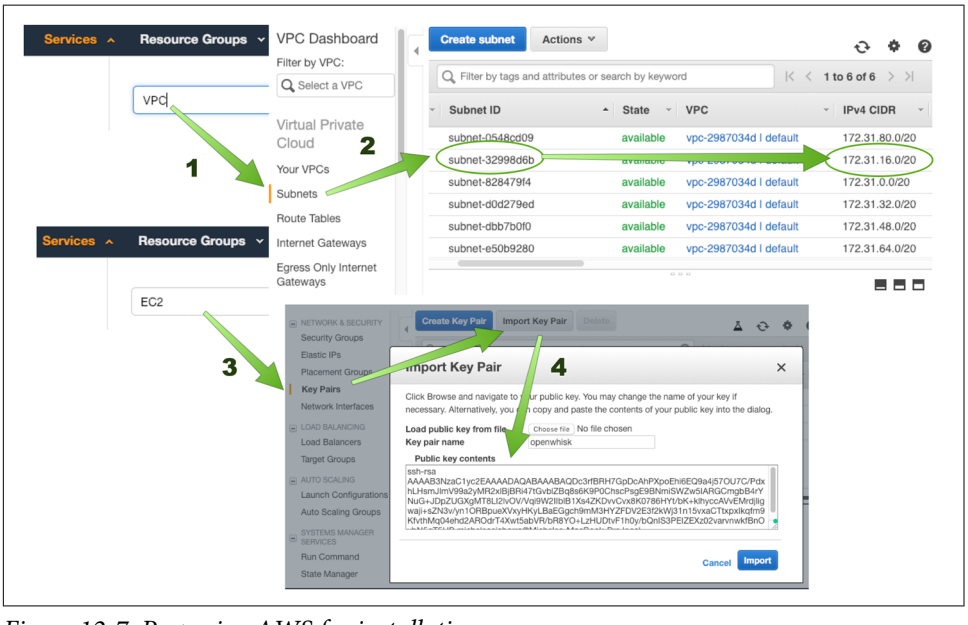


Figure 12-7. Preparing AWS for installation

If you do not have one already, create a private key with the command `ssh-keygen`. If you accept the defaults, your public key will be stored in the file *$HOME/.ssh/id_rsa.pub*. Let's proceed by locating the IP range (aka the subnet) of your VMs and installing the SSH key.

1. Click on "Services" in the menu bar and search for VPC.
2. Select "subnets", and note the available subnets; pick one and write down the name and the IPv4 CIDR.
3. Now click again on "Services" again and search for EC2, then select "Key Pairs."
4. Click on "Import Key Pairs" and in the text area paste your public key.

We refer to the name of the subnet as *<subnet-name>* and to the address range as *<subnet-cidr>*. Now we can launch the VM as shown in Figure 12-8.

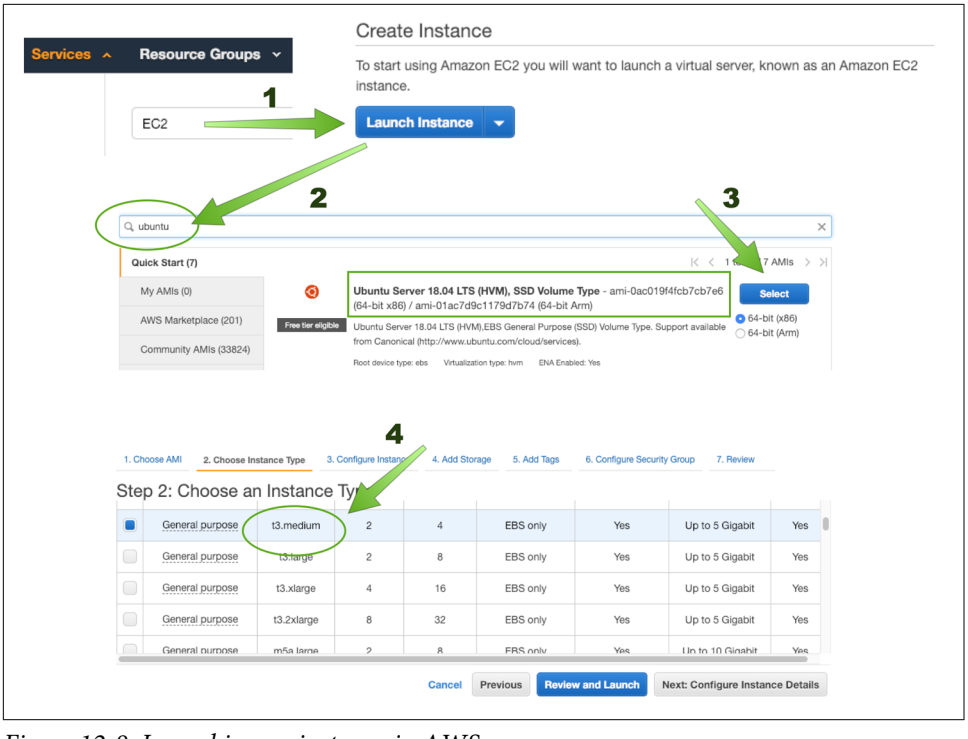


Figure 12-8. Launching an instance in AWS

Here are the steps:

1. Go to the EC2 Dashboard and click Launch Instance.
2. Search for "Ubuntu" and locate the Ubuntu Server 18.04 LTS image.
3. Pick the 64-bit version and click Select.
4. Select for the master at least a "t3.medium" instance (4 GB of memory), then click on "Next: Configure Instance Details."

It is important that you configure the VM carefully (as shown in Figure 12-9) because the details that follow are critical to making it work.

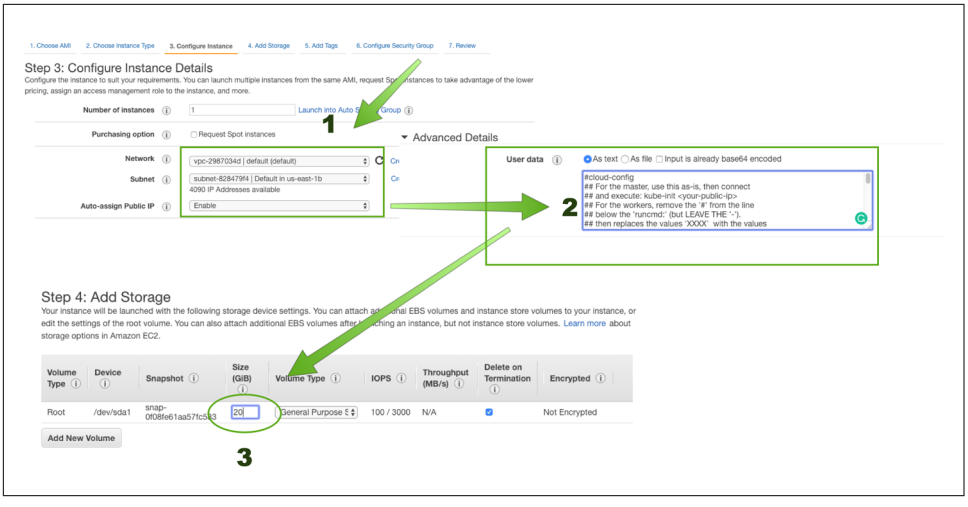


Figure 12-9. Configuring an instance in AWS

In the section marked **1** in Figure 12-9:

- Select one instance and use the default VPC.
- Pick the *<subnet-name>* you noted in the previous step (this is important!).
- Select Enable in the "Auto-assign Public IP" drop-down.

Now scroll down to Advanced Details and open this section; it will show a text area to put user data (marked as **2** in Figure 12-9). Select the "As text" option and paste in the `cloud-init` script. Then click Next: Add Storage.

For the VM (**3** in Figure 12-9), I recommend at least 20 GB of space. You can now skip the next step, clicking Next: Add Tags, and then go on to Next: Configure Security Group.

Carefully set the security constraints (otherwise the cluster won't work!), as shown in Figure 12-10.

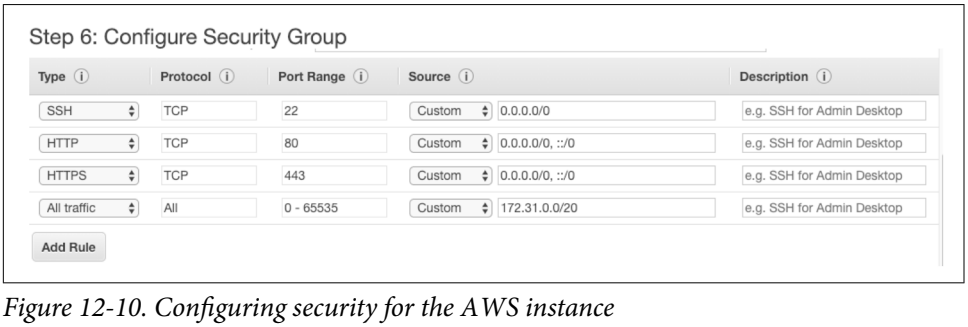


Figure 12-10. Configuring security for the AWS instance

Leave open the default port 22, so you can access the instance with SSH. You also need the master to expose to anyone the web ports, namely ports 80 (HTTP) and 443 (HTTPS). You also need to allow all the instances to talk to each other. To do this, you need to add a rule enabling "All traffic" using as the source the `<subnet-cidr>` address you noted earlier.

You are going to create a few instances that will talk to each other (because Kubernetes requires this). To do so, it is critical you enable traffic among the various instances of the Kubernetes cluster.

You are now ready to launch the master instance. When launching, remember to select the key you uploaded earlier. Afterward, the AWS console will show a list of VMs. Select the one you just started, and in the console you will see the IPv4 `<public-ip>` of the instance you have created.

Now you can log in to the VM using the command `ssh ubuntu@<public-ip>`. If you recall the discussion in "Architecture of a Kubernetes Cloud Deployment" on page 327, now you are in the case of a network with private IPs.

Type `cloud-init status --wait`, and wait until until the command terminates. Generally at the end it will disconnect your SSH connection. Reconnect and create the cluster with the command `sudo kube-init <public-ip>`, waiting until the cluster is ready.

At the end of the procedure, take note of the command starting with `kubeadm join` for the case of a private network.

You can now repeat the configuration, creating more instances (at least 2). The procedure is the same as described for creating the master, except in the user data text area you have to edit the `cloud-init` script, adding after `#cloud-config` the two lines shown in the output of `kube-init`.

Wait until the cluster is ready, monitoring the status with `sudo watch kubectl get nodes`, then complete the initialization with `sudo kube-provision <public-dns> <your-email-address>`.

Installing Kubernetes on a Bare Metal Server

Now let's see how to install Kubernetes on a bare metal server running a Linux-based operating system. We also leverage here the `cloud-init` installation we covered in the previous section. The procedure is similar to the cloud-based one, but it is more complicated because you need to prepare disk images and configure networks to launch the required VMs on your server.

As a prerequisite, you need a physical server with a reasonable amount of memory (I recommend at least 16 GB) where you've already installed a Linux distribution based on Ubuntu, Red Hat, or some other flavors. Here we use a Ubuntu server, but the procedure can be easily adapted to others (mostly by changing the names of the packages to install).

We also use the `libvirt` package to manage the VMs; it is available in all major Linux distributions. I'll use the command line here to describe these steps, but Kubernetes can also be installed using a graphical user interface (like virt-manager).

This section discusses installation on a single server, but Kubernetes can be run on VMs running on different servers as long as there is network connectivity among them. This is normally called a "private cloud" setup. Generally, private clouds also have a user interface. If you have a private cloud, see "Generic Procedure for Installing Kubernetes with cloud-init" on page 329 for instructions on how to create your Kubernetes cluster with your cloud management interface.

Collecting the Required Software

To build VMs in a server, you need to install:

- A VM manager (`libvirt-bin`)
- A VM installer (`virtinst`)
- Utilities to build ISO images (`cloud-utils`)
- A port redirector (`rinetd`)

You have to perform all the steps as the root user. You need a directory under */var/lib/libvirt* (e.g., */var/lib/libvirt/kube*) for all the configuration files, and you'll need to download the Ubuntu cloud image and our `cloud-init` script. Log in as root and type the following commands:

```
$ apt-get -y upgrade
$ sudo apt-get -y install libvirt-bin virtinst cloud-utils          ❶
$ mkdir /var/lib/libvirt/kube                                      ❷
$ cd /var/lib/libvirt/kube
$ curl -L https://learning-apache-openwhisk.github.io\             ❸
  /chapter12-deploy/user-data.txt \
  >user-data.txt
$ curl -L \                                                        ❹
  https://cloud-images.ubuntu.com/\
  xenial/current/xenial-server-cloudimg-amd64-disk1.img \
  >base.img
```

❶ Install the required software.

❷ Create the working directory.

❸ Download the `cloud-init` script.

❹ Download the Ubuntu cloud image on our server.

Network Configuration

It is mandatory that you enable IP forwarding to communicate with your VMs. Edit the file */etc/sysctl.conf* and include this line, if it's not present:

```
net.ipv4.ip_forward=1
```

If you change it, execute `sysctl -p` to enable the change now. This change will be picked up automatically at the next reboot.

You also need to open ports 80, 443, and 6443. The actual command to use depends on the firewall software you are using. If you are using the package `ufw` (uncomplicated firewall), execute:

```
$ sudo ufw allow http/tcp
$ sudo ufw allow https/tcp
$ sudo ufw allow 6443/tcp
```

Next, you need to set up port forwarding from the server to the master VM. In particular, you need to forward the ports 80, 443, and 6443 to reach the master. You can do this with the `rinetd` server. Add to the file */etc/rinet.conf* three lines in this format:

```
<server-ip> 80 <master-ip> 80
<server-ip> 443 <master-ip> 443
<server-ip> 6443 <master-ip> 443
```

Let's see how to find the actual values to use with an example. The *`<master-ip>`* depends on the `id` you will pick for the server, as described next. The `id` is used as the last part of the internal IP address of the VMs. We explain the `id` in detail later, but must be two digits. By default, `libvirt` creates an internal network at 192.168.122. Hence, if you pick `id=10` for your master, then its IP is going to be 192.168.122.10. Now read your server's IP address:

```
$ sudo hostname -i
123.45.67.8                                                    ❶
```

❶ Your actual IP will be different.

In this case, you would add the following lines:

```
123.45.67.8 80 192.168.122.10 80
123.45.67.8 443 192.168.122.10 443
123.45.67.8 6443 192.168.122.10 6443
```

Scripts for the Installation

Since there are many operations you have to perform more than once, scripts can be used. The first script, *build.sh*, helps to build images. It creates a disk image for the VM, copying and resizing the base image you downloaded. The cloud image requires `cloud-init` user data in an ISO image, so it also builds such an image with the data for the initialization:

```
ID=${1:?id}
SZ=${2:?size}
INIT="${3:-}"
UD=user-data-$ID.txt
cp base.img node$ID.img                                    ❶
qemu-img resize node$ID.img ${SZ}G                         ❷
cp user-data.txt $UD                                       ❸
echo "hostname: node$ID">>$UD                              ❹
echo "runcmd:" >>$UD                                       ❺
test -z "$INIT" || echo "- $INIT">>$UD
cloud-localds node$ID.iso $UD                              ❻
```

❶ Copy the base image in a new disk image.

❷ Resize the disk image to ensure it has the requested size.

❸ Copy the base *user-data.txt* script to customize it.

❹ Specify a different hostname for each image.

❺ Add where required an additional initialization command.

❻ Build the ISO image with the `cloud-init` user data script.

Next is a *boot.sh* script that can launch the image. The script first configures a new entry in the DHCP server to assign an IP to the VM, then creates the VMs, specifying all the parameters (memory, CPU, network interface, disk, and ISO):

```
ID=${1:?id}
MEM=${2:?memory}
OPT=${3:-}
ENTRY="<host name='node$ID' \                              ❶
ip='192.168.122.$ID' \
mac='52:54:00:92:68:$ID'/>"
virsh net-update default \                                 ❷
add ip-dhcp-host "$ENTRY" --live --config
virt-install \                                             ❸
```

```
--name node$ID \                                                    ❹
--ram $(expr $MEM \* 1024) \
--vcpu 2 \
--disk path=$PWD/node$ID.img \                                      ❺
--disk path=$PWD/node$ID.iso,device=cdrom \
--os-type linux \
--os-variant ubuntu16.04 \
--network bridge=virbr0,mac=52:54:00:92:68:$ID \                    ❻
--graphics none \                                                   ❼
--console pty,target_type=serial \
$OPT
```

❶ Define IP and MAC address for the VM.

❷ Create the entry in the DHCP server.

❸ Start the VM.

❹ Specify the name, memory, and number of virtual CPUs.

❺ Specify the disk and cdrom image.

❻ Specify the MAC address of the network connection.

❼ Disable graphics and enable a serial console.

Creating the Cluster

Now you are ready to build your cluster. Before starting, you need to know your *<server-address>*—either the IP address or the DNS name you use to reach your server. This is generally the address you use when you log in with SSH. Sometimes you have both a private IP address and a public IP address. In this case, the *<server-address>* is generally the public IP.

Now, as a first step, edit the *user-data.txt* script, adding a password for the VMs you are going to create. Ensure that you have uncommented the following lines, replacing `Change-me!` with your password:

```
chpasswd:
  list:
   - ubuntu:Change-me!
```

Now prepare the master. You have to choose an `id` in the range 10–99 (must be exactly two digits). The `id` will be used as the last digit of your IP address and as the name of your VM. The `id` is also used internally in the MAC address of the virtual machine, as you cannot assign an IP directly when creating a VM, but you can associate an IP with a MAC address in the DHCP server.

If you choose `10` as your `id`, you will get the IP `192.168.122.10` for your VM and the name `node10`. You can build an image specifying the `id` and the size of the disk image in GB. After building the VM you can boot it, specifying again the `id` and the amount of memory in GB to assign to it. For example, with `id=10`, to build a disk image of 20 GB and boot a VM with 4 GB of RAM you run the commands:

```
$ sh build.sh 10 20
Image resized
$ boot.sh 10 4
... output omitted ...
```

Now the VM boots and then initializes. Note that in your terminal, if you use these two commands, you will see output for the VM, not your server. You should wait until you see the message:

```
Reached target Cloud-init target.
```

You can now log in to the VM. Press enter and log in as *ubuntu* with your password. First, you will be asked to change your password. Then, you can execute the command to initialize the master:

```
sudo kube-init <server-address>
```

This command initializes Kubernetes. While the system software has been installed, configuration files and Docker images have to be downloaded to actually install Kubernetes. When this is done, you'll see two snippets to add to the user data to join the cluster in the format:

```
runcmd:
- kubeadm join <various-information>                    ❶
```

❶ You only need the `kubeadm` command.

The first snippet is the one you need because the cluster created with `libvirt` is equivalent to a private network as described in "Architecture of a Kubernetes Cloud Deployment" on page 327. Also, you *do not need* the whole snippet, only the command part starting with `kubeadm join`.

Note that you are currently logged into the VM. You need to exit from it and come back to the server. You can do this by pressing Ctrl-].

Once you are back in the server, save in a variable the `INIT` command to join the clusters.

```
export INIT="kubeadm join <various-informations>"
```

Now you have to build the workers. The absolute minimum is two, but it is better to have three workers. To identify the workers, pick a few more `ids` in the range 10–99 that you have not used before.

Now you can build three images, using again the *build.sh* script. As you may remember, the first parameter is the `id`, and the second is the size in GB. The script also accepts a third parameter, a command added to the `cloud-init` user data script. You can use it to instruct the workers to join the Kubernetes cluster.

You can now build the worker images with:

```
$ for i in 11 12 13 ; do sh build.sh $i 20 "$INIT" ; done
Image resized.
Image resized.
Image resized.
```

Once the images are ready, you can boot all of them with the *boot.sh* script. Again, the first parameter is the `id` and the second is the size of the memory in GB, but it also accepts additional parameters .

Since you do not need to interact with the workers while creating them, use this parameter to pass the option `--noautoconsole` to start the VMs in the background:

```
$ for i in 11 12 13 ; do sh boot.sh $i 4  "--noautoconsole" ; done
... output omitted ...
```

You can now come back to the master VM with the command `virsh console node10`. Execute the command `sudo watch kubectl get nodes` and wait until all the nodes are ready.

Now, stop the command with Ctrl-C. The next step is to provision the cluster. The command to use depends on if your server has a public IP and if it is connected to the internet. If it is not, you cannot get an SSL certificate, so just execute `sudo kube-provision`.

If instead your server has a public IP address, you can provision it and also get an SSL certificate using the command `sudo kube-provision` *`<server-address>`* *`<email-address>`*, where *`<server-address>`* is the public IP address and *`<email-address>`* is your email address.

If you do not have a public IP address and you cannot get an SSL certificate, you will have to use the command `wsk` with the `-i` switch to allow insecure access.

Installing OpenWhisk

Now that you have your Kubernetes cluster up and running, let's install OpenWhisk using Helm. Helm uses a package descriptor called a chart, which you can download from GitHub with the command:

```
$ git clone https://github.com/apache/incubator-openwhisk-deploy-kube
```

The chart is in flux and changes frequently. Check the repository to see updated documentation. A copy of the chart for this chapter is archived in the book's GitHub repository.

Configuring Kubectl

Kubernetes does not use passwords, but cryptographic certificates. To use Helm and `kubectl` to install OpenWhisk, you'll need a configuration file containing those certificates. You can find this file on the master node of your Kubernetes cluster in */etc/kubernetes/admin.conf*. You need to copy this file locally in *$HOME/.kube/config* to access the Kubernetes cluster.

If you are using Docker Desktop for Mac or Windows, this file will be configured automatically. You can make sure everything is working by verifying that the following command returns this output:

```
$ kubectl get nodes
NAME                 STATUS   ROLES    AGE   VERSION
docker-for-desktop   Ready    master   10d   v1.10.11
```

If you instead want to access a remote Kubernetes cluster you've built, you can download the certificate from your cluster to your local workstation using the following command:

```
$ ssh <user>@<master-ip> \
  sudo cat /etc/kubernetes/admin.conf >openwhisk.config
```

The *<user>*, *<master-ip>*, and credentials you need to access the cluster will depend on your configuration. If you are on the cloud, you can use the same values used to log in to the master.

If you instead are installing on your bare metal server, you may need to log in to your server and use the IP 192.168.122.*<id>* to access the VM, copy your certificate on the server, and then copy it again on your workstation.

Now, check if you can access the server cluster. Note you have to use the *<external-ip>* you provided when installing the cluster. For the cloud, the *<external-ip>* is the IP you used to connect to the master VM. For the bare metal installation, it is the IP of your server.

Use the following command to check if you can see the nodes:

```
$ kubectl --kubeconfig=openwhisk.config \
 --server=https://<external-ip>:6443 get nodes
NAME     STATUS   ROLES    AGE   VERSION
node10   Ready    master   22h   v1.11.6
node11   Ready    <none>   22h   v1.11.6
node12   Ready    <none>   22h   v1.11.6
node13   Ready    <none>   22h   v1.11.6
```

In the certificate file there is only the value to connect using the internal IP, but if you connect from your local machine, you are using a different IP and you have to use the parameter `--server` to use that certificate.

To avoid specifying the server when using the command line, you can edit the file *openwhisk.config*, locate the line with `server: https://<internal-ip>:6443`, and replace it with `server: https://<external-ip>:6443`. To avoid to specifying the path of your *openwhisk.config*, you can either copy the file in *$HOME/.kube/config* or set the environment variable `KUBECONFIG` to point to your configuration file.

If you also have Docker Desktop, copying *openwhisk.conf* over *$HOME/.kube/config* will overwrite the existing configuration and you will be unable to access the Docker for Desktop Kubernetes cluster. Ensure that you do a backup copy before overwriting it.

Configuring Helm

I assume now that you have retrieved the Kubernetes *admin.conf* file, edited it to point to the external IP of your server, and copied it as the default configuration (or equivalently, you have set the `KUBECONFIG` environment variable). You are now ready to use Helm.

Helm requires two steps before you can use it: initializing and obtaining permission. The initialization installs a server-side component in Kubernetes and creates some configuration files in your local workstation. You initialize Helm simply with:

```
$ helm init
...
$HELM_HOME has been configured at /Users/michelesciabarra/.helm.

Tiller (the Helm server-side component) has been installed into your Kubernetes
Cluster.
...
Happy Helming!
```

Helm is now up and running, but it requires some additional permissions to be allowed to do its work. So, you must execute the following command:

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

clusterrolebinding.rbac.authorization.k8s.io "tiller-cluster-admin" created
```

Congratulations! You are now ready to install OpenWhisk using Helm.

Installing in Docker Desktop

In this section I'll show you how to use the Helm chart to install Kubernetes locally. This deployment is intended for development only, since there is only one node, and works without any certificate in insecure mode only.

To install OpenWhisk you need to:

1. Label the master to be able to execute the invoker.
2. Get the internal IP to use as a parameter.
3. Write a parameter file with the appropriate values.
4. Deploy the chart using the parameter file.

Let's do these steps in order. (Before starting, ensure you are talking to the right cluster, and the configuration works properly, as described in "Configuring Kubectl" on page 346.)

Label the master

In Kubernetes you can add labels to various resources, including nodes. The OpenWhisk Helm chart uses Kubernetes labels to decide which components to run in each node. Some labels are optional, but it is mandatory to configure which nodes will run actions.

In Docker Desktop, there is only one node, the master, so you can mark the node with the following command and then check the node labels:

```
$ kubectl label nodes --all openwhisk-role=invoker
node/docker-for-desktop labeled
$ kubectl get nodes --show-labels
NAME                 STATUS   ROLES    AGE   VERSION    LABELS
docker-for-desktop   Ready    master   10d   v1.10.11   beta.kubernetes.io/\
arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=docker-for-desktop,\
node-role.kubernetes.io/master=,openwhisk-role=invoker
```

Get the internal IP

In Docker Desktop, Kubernetes runs in an internal VM with its own IP address. You need to know this IP and communicate it to the Helm chart. You can read the value with the following command:

```
$ kubectl get nodes -o=jsonpath='{.items[0].status.addresses[0].address}'
192.168.53.3                                                                    ❶
```

❶ Your actual value may be different.

Write a parameter file

Now that you have gathered all the informations required to invoke the helm chart, create a file called *owlocal.yaml* in the folder *incubator-openwhisk-deploy-kube* with the following contents, changing the IP address after `apiHostname`:

```
whisk:
    ingress:
        type: NodePort
        apiHostName: 192.168.53.3                                               ❶
        apiHostPort: 31001
nginx:
    httpsNodePort: 31001
```

❶ Change here the sample IP to the actual IP you found in the previous step.

Deploy the chart

You are ready to deploy. You have to invoke the chart (located in the folder *helm/openwhisk*), also specifying a name and a namespace for the deployment and feeding the parameters:

```
$ cd incubator-openwhisk-deploy-kube
$ helm install \                                                                ❶
   --name openwhisk \                                                           ❷
   --namespace ow \                                                             ❸
   -f owlocal.yaml \                                                            ❹
   ./helm/openwhisk                                                             ❺
```

❶ Invoke the `install` command for Helm.

❷ Name of the deployment.

❸ Namespace of the deployment.

❹ Parameters of the deployment.

❺ Helm chart of the deployment.

You have to use explicitly the name `./` to say to Helm that you want to use a local chart. Do not use `helm/openwhisk`, because it will search for it on the internet.

Now execute the following command to monitor the installation:

```
$ watch kubectl --namespace ow get pod
```

Then wait until the pod with the name starting with `openwhisk-install-packages` completes. It can take some time since there are many initializations to complete.

You can now proceed to "Configuring the OpenWhisk Command-Line Interface" on page 353 to configure the `wsk` CLI to access OpenWhisk.

Installing in the Kubernetes Cluster

In this section, I'll show you how to use the Helm chart to install OpenWhisk in a Kubernetes cluster you have built either in the cloud or on a local server.

The steps are as follows:

1. Label the workers to be able to execute the invoker.
2. Generate some security credentials.
3. Write a parameter file with the appropriate values.
4. Deploy the chart using the parameter file.

We'll go over each of these steps in detail in the following subsections. Before starting, ensure you are talking to the right cluster and have configured access to Kubernetes properly, as described in "Configuring Kubectl" on page 346.

Label the workers

In Kubernetes you can add labels to various resources, including nodes. The OpenWhisk Helm chart uses Kubernetes labels to decide which components to run in each node. Those configurations are mostly optional, but it is mandatory to label which nodes will run actions.

Let's assume we want to run actions in all the worker nodes of our Kubernetes cluster. To do this we can first enumerate the workers, storing the value in a variable, and then label each node with the label `openwhisk-role=invoker`, as follows:

```
$ WORKERS=$(kubectl get nodes -l \!node-role.kubernetes.io/master \
  -o jsonpath='{..name}')
$ for i in $WORKERS ; do \
$ kubectl label node $i openwhisk-role=invoker ; done
node/kube1 labeled
node/kube2 labeled
node/kube3 labeled
```

Generate credentials

You are now going to deploy a cluster that can be used in production and exposed to the public, so you cannot use default credentials; you have to provide secret values to prevent unauthorized access to your cluster.

OpenWhisk puts authentication keys in this format:

```
23bc46b1-71f6-4ed5-8c54-816aa4f8c502:\
123zO3xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyGVCGuMDGIwP
```

This key is not a random value: it is the default key used for the `guest` namespace if you do not specify other values.

The first part (the user id) is a UUID version 4, while the second part (separated by a `:`) is a random string of 64 alphanumeric characters. You can generate passwords with this Python 3 script:

```
import uuid,random,string
user = uuid.uuid4()
chars =  string.ascii_letters+string.digits
pswd = "".join(random.choices(chars,k=64))
print("%s:%s" % (user, pswd))
```

Run it twice to generate two credentials—one for the system and another for the guest user.

Write a parameter file

Now you are ready to write your configuration file. This file must include the:

- *<external-address>* of your Kubernetes cluster
- *<system-credentials>* and *<guest-credentials>* for OpenWhisk
- *<database-user>* and *<database-password>* for the CouchDB database

Write the configuration file as follows, replacing the appropriate values:

```
whisk:
  ingress:
     type: Standard
     domain: <external-address>
     apiHostName: <external-address>
     apiHostPort: 443
  auth:
    system: "<system-credentials>"
    guest: "<guest-credentials>"
db:
```

```
  auth:
    username: "<database-username>"
    password: "<database-password>"
```

Deploy the chart

You can now deploy the Helm chart:

```
$ cd incubator-openwhisk-deploy-kube
$ helm install \                                    ❶
   --name openwhisk \                               ❷
   --namespace ow \                                 ❸
   -f owcluster.yaml \                              ❹
   ./helm/openwhisk                                 ❺
```

❶ Invoke the `install` command for Helm.

❷ Name of the deployment.

❸ Namespace of the deployment.

❹ Parameters of the deployment.

❺ Helm chart of the deployment.

Now execute the command:

```
$ watch kubectl --namespace ow get pod
```

Then wait until the pod with the name starting with `openwhisk-install-packages` completes. In the following, there is an example of the output of a successful deployment:

```
$ sudo kubectl -n ow get po
NAME                                          READY   STATUS      RESTARTS   AGE
openwhisk-alarmprovider-556f977dc9-tpd7q      1/1     Running     0          3h
openwhisk-apigateway-589bf7d965-d7gkw         1/1     Running     0          3h
openwhisk-cloudantprovider-7659d5b855-l498t   1/1     Running     0          3h
openwhisk-controller-0                        1/1     Running     2          3h
openwhisk-couchdb-7f7f9479b6-rsklx            1/1     Running     0          3h
openwhisk-init-couchdb-gs4cj                  0/1     Completed   0          2h
openwhisk-invoker-5nl74                       1/1     Running     0          3h
openwhisk-invoker-pbjk9                       1/1     Running     0          3h
openwhisk-invoker-wgbgw                       1/1     Running     0          3h
openwhisk-kafka-0                             1/1     Running     0          3h
openwhisk-kafkaprovider-579b6ff6fd-79srl      1/1     Running     0          3h
openwhisk-nginx-76dc856998-xx7bc              1/1     Running     0          3h
openwhisk-redis-7cf6c88fd9-b7jnp              1/1     Running     0          3h
openwhisk-zookeeper-0                         1/1     Running     0          3h
```

Configuring the OpenWhisk Command-Line Interface

If you've followed along, you now have OpenWhisk installed. But to use it, you'll need to configure it at the command line with the API host and the authentication key. Let's see how to set those, first for a local deployment with Docker Desktop, then for internet deployment.

Configuring wsk Insecurely for Docker Desktop

For a local installation using Docker Desktop the API host is `localhost:31001`, while the default namespace is `guest`. You can find the default authentication key in the Helm chart deployment, under *helm/openwhisk/values.yaml*, in the line starting with `guest:`. The `wsk` command line has the `wsk property set` subcommand with the flags `--apihost` and `--auth` to set them.

The local installation does not deploy SSL authentication, so you have to use `wsk` with the `-i` (insecure) flag. It may be useful to set `alias wsk=wsk -i`. For example, you can configure the `wsk` command from the command line with:

```
$ cd incubator-openwhisk-deploy-kube
$ eval $(grep guest: helm/openwhisk/values.yaml \                    ❶
  | awk '{print "AUTH="$2}')
$ wsk -i property set --apihost http://localhost:31001
ok: whisk API host set to localhost:31001
$ wsk -i property set --auth $AUTH
ok: whisk auth set. Run 'wsk property get --auth' to see the new value.
```

❶ Extract the authentication key to set the `AUTH` variable.

If instead you want to configure your cluster, if you followed the recommendations in this chapter you know your API host (it is the public IP of your Kubernetes cluster) and your API key, since you generated it. The namespace is `default`.

Creating a New Namespace

Now, let's assume you have a deployment exposed to the public internet. As discussed earlier, you should have changed the authentication key so your `system` and `guest` users do not use the default values. You can then use the authentication key you have set for the `guest` namespace. However, you will likely want to create more namespaces.

OpenWhisk allows creating multiple namespaces using the `wskadmin` utility. In general, to access this utility, you need to have direct access to the database. In a Kubernetes deployment, the database is not exposed outside of the cluster, nor it is advisable to do so. The Helm chart deploys a component (a *pod* in Kubernetes parlance), configured to access the database, containing this utility with the environment

already configured to access the OpenWhisk database. If you have access to the Kubernetes cluster, you can get shell access to this pod to perform administrative tasks.

The name of the pod depends on the name and namespace of the Helm deployment you used. If you followed the examples in "Installing OpenWhisk" on page 345, the name of the deployment is `openwhisk` and the namespace is `ow`.

Under those assumptions, the command to access the `wskadmin` utility is:

```
$ kubectl exec -ti \                          ❶
  --namespace ow \                            ❷
  openwhisk-wskadmin \                        ❸
  -- bash                                     ❹
```

❶ Command to execute an interactive command in a pod.

❷ The namespace is the one where you deployed OpenWhisk.

❸ The pod name is `wskadmin` prefixed with the deployment name.

❹ Run a shell in the pod.

Once you have shell access, you can create a namespace. OpenWhisk, however, internally creates users then associates namespaces with them. So, you can create a user (for example, `devel`) and a namespace with the same name in a single step with the command:

```
# wskadmin user create -ns devel devel        ❶
aaaaa:BBBBBBBBB                               ❷
```

❶ Run this command in the shell you launched with `kubectl`.

❷ Simplified version; the real key is longer and with random characters.

The utility created the user, and you get in the output the authentication key to use for your new namespace. Take note of it.

You can now exit from the `kubectl` shell (just type `exit`) and configure `wsk` to use the new namespace with:

```
$ wsk property set \
  --apihost https://your.openwhisk \          ❶
  --auth aaaaa:BBBBBBBBB  \                   ❷
ok: whisk auth set. Run 'wsk property get --auth' to see the new value.
ok: whisk API host set to https://your.openwhisk
```

❶ Change to your actual API host.

❷ Change to the actual generated key.

Summary

After reading this chapter you should be able to install Kubernetes and OpenWhisk in your favorite environment.

We covered how to install Kubernetes as a single node using Docker Desktop and how to create a cluster in general. Then you saw examples of installation on Hetzner Cloud, AWS, and a private server. We also used the tool Helm to install OpenWhisk in those environments.

Conclusion

We are at the end of our journey here. You have seen a lot of different ways to use OpenWhisk, in JavaScript, Python, and Go.

You have also seen plenty of examples simple enough to use as starting points, including: a contact form, a CRUD application, a web chat application, and many useful examples of design patterns. You've explored CouchDB, Kafka, and even learned how to install Kubernetes.

Actually, the programming languages I picked are somewhat arbitrary, since the number of available languages for OpenWhisk is pretty high. Among the scripting languages we have are also PHP, Ruby, and Perl. You can also choose to use programming languages based on the Java Virtual Machine, like Java or Scala. And you can compile programming languages, from the veteran C and C++ to the more modern Swift and Rust.

This is the end of the book, but for you, it is just the beginning: you can now write your cloud-native applications using open source technologies, with the freedom to deploy them in any environment you want.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

About the Author

Michele Sciabarrà is a veteran of information technology and is currently CEO of *sciabarra.com*, a consultancy focused on Kubernetes and Serverless solutions. He's also a contributor to the Apache OpenWhisk project, most notably as the author of the high-performance ActionLoop runtime for Go, Swift, Rust, Java, and other programming languages.

Colophon

The animal on the cover of *Learning Apache OpenWhisk* is the armed hermit crab (*Pagurus armatus*), also known as the black-eyed hermit crab. There are over 1,100 species of hermit crab, and they're found in all kinds of environments, including fresh water, salt water, and dry land. The armed hermit crab inhabits the Pacific coast extending as far north as British Columbia, Canada.

Like all hermit crabs, the armed hermit crab lacks an exoskeleton on its long, soft abdomen, so must wear the abandoned shell of a mollusk, such as a sea snail or conch. At 43 mm in length (about 1.7 inches), the armed hermit crab is one of the largest species. In addition to its characteristic large, black, compound eyes, the armed hermit crab has spiny, red, orange, and white striped legs. It feeds on the eggs of other sea creatures, such as lobsters and fish, as well as zooplankton and carrion.

Some people keep hermit crabs in aquariums as pets, but the armed hermit crab can't survive in captivity.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover image is a color illustration by Karen Montgomery, based on a black and white engraving from *Animal Life in the Sea and on the Land*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.