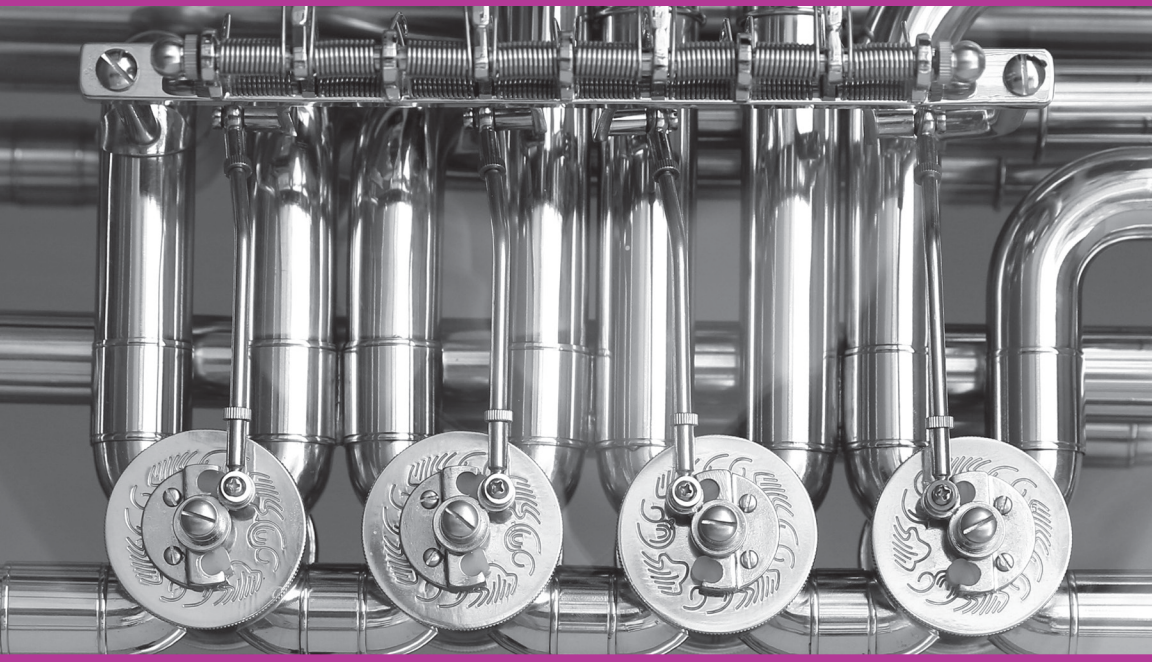
O'REILLY®

Compliments of IBM

Developing Serverless Applications

A Practical Introduction with Apache OpenWhisk



Raymond Camden

Control costs, autoscale, and integrate systems across clouds.

| IBM Cloud Functions

Based on Apache OpenWhisk, IBM Cloud Functions provides an ecosystem in which developers can share and use code across environments.

Run actions thousands of times in a second or once a week. Action instances scale to meet exact demand, then disappear.

It's simple. Pay only for exact times your actions run, down to one-tenth of a second: no memory, no cost.

Trigger your actions from events in your favorite cloud services, or directly from your apps via REST APIs.

Develop actions in your favorite language: Java, Swift, JS/Node.JS, PHP, Python. You can also use any code in a Docker container as an action.

✓ Focus on building and running apps, not maintaining servers

✓ Quickly bind AI services to your action sequences

✓ Choose a cloud environment with open source tools

IBM Cloud Functions

https://ibm.co/CloudFunctions

Developing Serverless Applications

A Practical Introduction with Apache OpenWhisk

Raymond Camden

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Developing Serverless Applications
by Raymond Camden

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/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Brian Foster and Virginia Wilson
Production Editor: Shiny Kalapurakkel
Copyeditor: Matthew Burgoyne
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest
Tech Reviewers: Mike Roberts, John Chapin, Brian Rinaldi, Jess Males

October 2017: First Edition

Revision History for the First Edition
2017-10-06: First Release

978-1-491-99622-5

[LSI]

Table of Contents

CHAPTER 1

Introduction

This book aims to help you enter the exciting new world of serverless development. By the end of the book, you'll know how to use Apache OpenWhisk to write serverless functions and how to call them from applications. You'll see how to create powerful new services by combining existing actions into sequences, as well as how they can be automated with triggers.

What Is Serverless?

The technical industry is rife with terms that sometimes bear little to no resemblance to what they supposedly represent. Serverless is the latest example of this. Many people object to the term because servers are *always* involved with anything hosted on the internet. Of course, if we can get over the fact that the cloud isn't literally a few thousand feet over our heads, we can move to accept this new term as well.

It's best to think of serverless as not the lack of servers, but the lack of server management. Serverless is best described as the ability to deploy your apps without worrying about provisioning and managing a server. This could be seen as a natural outgrowth of the cloud and virtual servers.

Virtual servers greatly improved our lives by allowing developers to add (and remove) servers with the click of a button. Suddenly, you didn't have to ask permission, or wait around, to put your application online. As awesome as this was, it came with some drawbacks.

Servers require maintenance. They have to be locked down. They have to be set up with the appropriate amount of RAM and disk space. They also have to be running to actually do anything. While that's obvious, it's easy to forget that you're paying for a server to run all day, every day, even when the app in question isn't actually being used.

This is where serverless truly shines. Instead of provisioning an entire server, you can take the logic that comprises your application and simply deploy it as is. You can think of this as "Function as a Service." This greatly simplifies the process of getting what you need online and ready to be used. You no longer need to worry about setting up a web server or figuring out how to route a particular request to a particular piece of logic. Instead, you deploy that logic (a function) and the serverless provider handles everything else.

Best of all, your costs can go way down. Instead of a server running constantly, the serverless platform fires up your code when requested. When your code isn't being used, nothing is running. You pay for your usage and nothing more! While every platform has it's own pricing model, your costs in general will be far less.

If you think this sounds compelling, you aren't alone. Many of the biggest tech companies now offer serverless as a product offering. Amazon led the way with AWS Lambda, the oldest and most mature serverless platform. Google has Cloud Functions, and Microsoft has Azure Functions. In this book, we'll be focusing on IBM Cloud Functions, a serverless solution created by IBM based on Apache OpenWhisk.

Serverless Use Cases

It is (most likely) simpler to work with serverless and certainly cheaper, but what are some use cases where serverless shines? Here are just a few examples.

APIs for CRUD
: CRUD, or Create/Read/Update/Delete, refers to how people typically work with content. Given you have a set of data, your app may need to perform CRUD as part of its functionality. In the past, APIs for this would be built in a server-side application and handle passing calls back and forth between a frontend and a backend storage system. Serverless is great for these simple

proxies by reducing the amount of overhead you need to support the APIs.

Rewriting APIs
: One of the cooler things you can do with serverless is build an API that rewrites another API. Imagine an API that only returns XML. You could use serverless to build a proxy that transforms it to JSON. Imagine an API that returns useful information, but also a lot of data your clients don't need. You could use your API to simply remove the unwanted data.

Slow background work
: Anything that is slow (comparatively) will typically be handled by a process running in the background on a scheduled basis. Examples of this include image processing (such as optimizing and resizing), log processing, or simply handling uploaded files. Serverless is a great way to build these background utilities.

Who Is This Book For?

This book is for the developer who works on the backend, creating services to support frontend clients. Those clients could be basic websites or more modern progressive web apps. They can also be mobile apps written using hybrid technologies like Apache Cordova, or fully native apps written in Java or Swift. This book is for anyone who has had to set up an environment, a web server, and other tools just to build an API, and would love to see a much simpler way of standing up services.

Apache OpenWhisk supports multiple languages, but this book focuses on JavaScript for working with serverless. You do not need to be a JavaScript expert in order to work with the examples. If you know nothing at all about JavaScript and want to quickly get up to speed, or just want to brush up on your knowledge, I highly recommend bookmarking the MDN Web Docs, seen in Figure 1-1. Formally known as "MozDevNet," this site contains both guides and reference materials to all things web related.

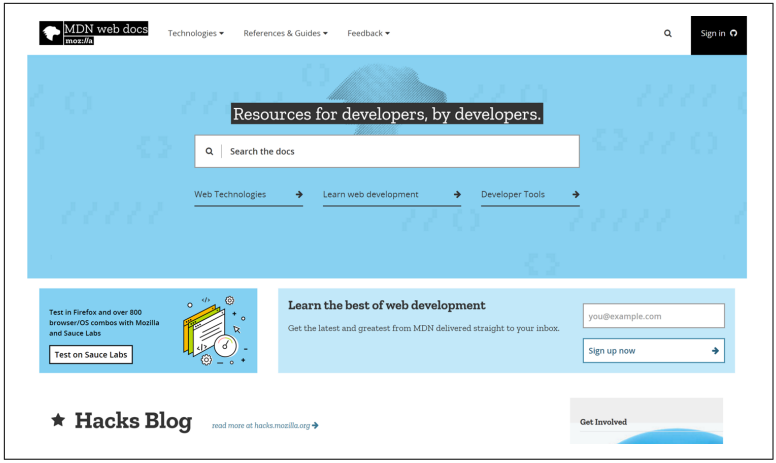


Figure 1-1. MDN Web Docs website

There are absolutely no editor or web browser requirements to complete this book. Use the editor and browser that works best for your development. I used Visual Studio Code and highly recommend it.

Getting the Source

You can find the complete source for the code samples in this book on GitHub at *https://github.com/cfjedimaster/developing-serverless-applications-code*. If you have a Git client installed on your system, simply pull a copy of the repository. If you don't, use the Clone or download button to get a zip of the files.

CHAPTER 2

OpenWhisk Basics

One of the best benefits of OpenWhisk is how easy it is to get started. You'll learn what IBM Cloud Functions based on OpenWhisk and IBM Cloud (Bluemix) are and how they relate. Then, you'll learn how to get started. That process is divided into two parts. First, I'll walk you through registering with IBM Bluemix. While optional, this provides the simplest way to test your OpenWhisk actions. Second, I'll show you how to install the command line interface for OpenWhisk, how to confirm it worked, and what the upgrade process is like.

Introduction to Apache OpenWhisk

Apache OpenWhisk is a new, open source project under the Apache umbrella (see Figure 2-1 for a screenshot of its website). Many people tend to think of Apache as "the web server people," but the organization is responsible for many open source projects.

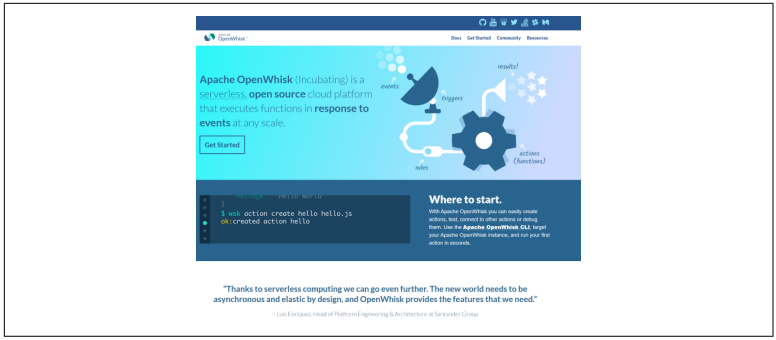


Figure 2-1. The Apache OpenWhisk website

Apache OpenWhisk has some pretty big supporters, including IBM, Adobe, and most recently, Red Hat.

As an open source project, anyone and everyone can participate in the platform. There is both a public mailing list as well as GitHub repositories where bugs can be reported (and worked on). There is also a Slack organization where people can ask technical questions. You can sign up for that Slack group and, once registered, you can chat with other developers.

OpenWhisk on IBM Cloud (Bluemix)

As one of the biggest supporters behind OpenWhisk, IBM has made it easy for developers to use the serverless platform in IBM Cloud Functions, the managed OpenWhisk implementation on the IBM Cloud (Bluemix). IBM Cloud is a Platform as a Service (PaaS) solution that handles things like NodeJS hosting, integration with Watson, and OpenWhisk (see Figure 2-2 for a screenshot of its website). Because IBM Cloud is easy to use, this book will focus on using that platform for deploying and hosting your serverless code. To be clear, this is *optional* and 100% *not required* to use Apache OpenWhisk. IBM Cloud Functions has an incredibly generous free tier, so you will not have to pay a dime to run through the various exercises and examples in this book. In this context, we will use IBM Cloud Functions and OpenWhisk interchangeably.

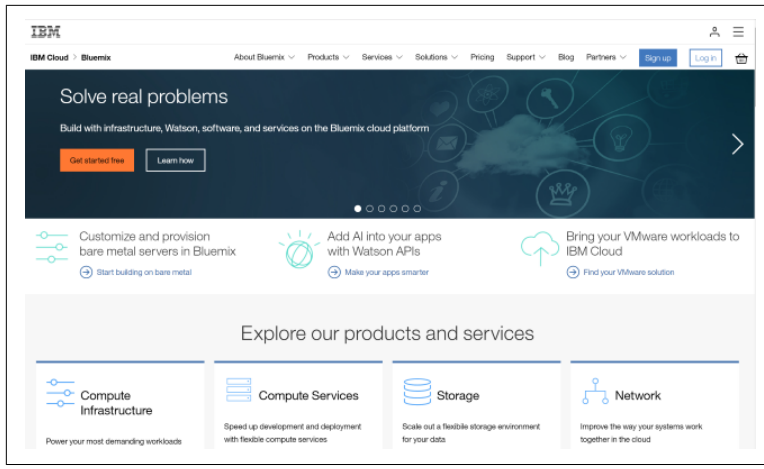


Figure 2-2. The IBM Cloud (Bluemix) website

IBM's website refers to OpenWhisk as Cloud Functions. This is the name IBM uses to refer to OpenWhisk running in its own offerings.

In the next chapter, I'll walk you through how to use OpenWhisk with Bluemix. However, note that the core of your OpenWhisk activity will be the same no matter where you end up deploying your code.

Registering for IBM Cloud (Bluemix)

If you don't currently have a Bluemix account, open the registration page and sign up (see Figure 2-3).

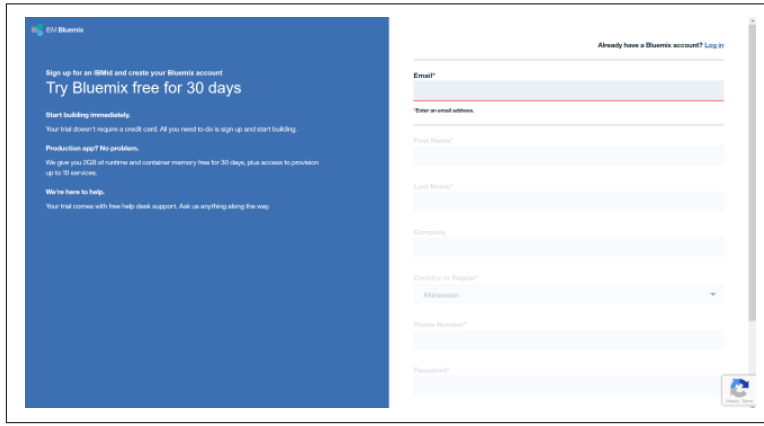


Figure 2-3. Registration page for Bluemix

This should be like any other registration page you've encountered before. Note that you are *not* asked for a credit card. The registration page mentions a 30-day free trial. While that's true, you can still use IBM Cloud Functions for free after your 30 days. It has a free tier (as does most of Bluemix), and as long as you don't share your serverless APIs with a few million of your friends, you shouldn't have anything to worry about.

If you've already registered for Bluemix in the past, you don't need to do anything special. Just log in. Then, skip to the next section.

After signing up, you'll need to do an email verification. Sign in, and you'll be asked a few questions. The first is to simply specify an organization (see Figure 2-4). Go ahead and use one of the suggested values. This really isn't that important.

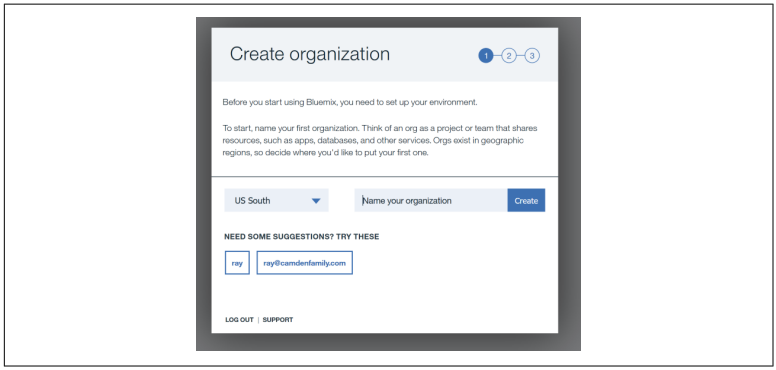


Figure 2-4. Picking an organization

Next, you'll be asked to create a "space." Think of this as a folder or project for your work. If you do a lot with Bluemix, you may have many of these to organize your work. But as before, just select one of the suggested values (see Figure 2-5).

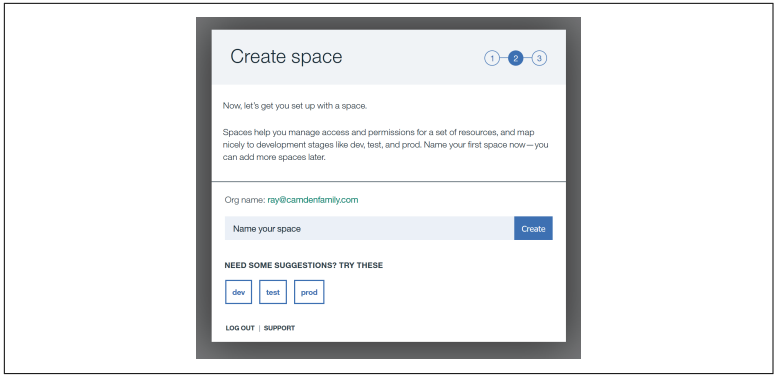


Figure 2-5. Picking a space

Congratulations, you've registered for Bluemix! In the next section, you'll learn how to get the OpenWhisk command line interface, as well as how to configure it.

Getting the OpenWhisk CLI

The Bluemix dashboard will be pretty empty for new users. For existing users, you'll see what you normally do here. To get to OpenWhisk, click the hamburger menu in the upper lefthand side, as shown in Figure 2-6.

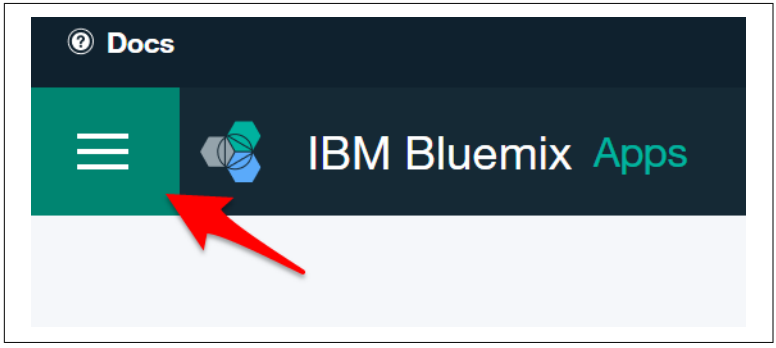


Figure 2-6. Opening the Bluemix menu

Then, select Functions from the menu. This brings you directly to the "Getting Started" experience. Here, you'll find documentation, links to develop and monitor OpenWhisk activity, and the API portal. For now, begin by clicking the Download Cloud Functions CLI button, as shown in Figure 2-7.

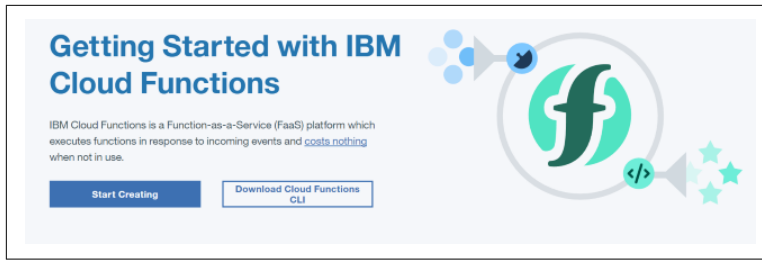


Figure 2-7. The Getting Started page

This page walks you through setting up the Bluemix CLI, but you can skip this as this book is focused on OpenWhisk development only. At the bottom of the page, simply click the last link: "Looking for the wsk CLI?" The next page (as shown in Figure 2-8) includes a bunch of critical information, so let's cover it bit by bit.

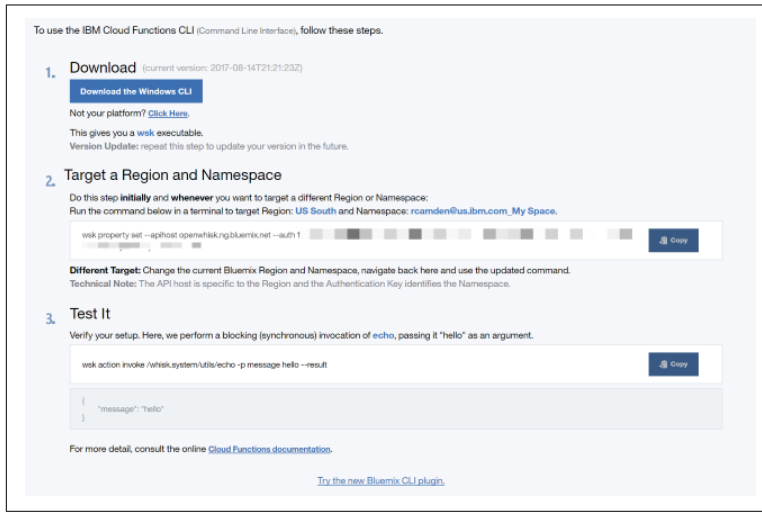


Figure 2-8. The Download page

First and foremost, note that the web page does a bit of sniffing to determine what operating system you're using. For most folks, this will be fine. However, if you need another operating system, you would click the link marked "Click here." For Windows users using the new Windows Subsystem for Linux (WSL), this is what you would do to get the Ubuntu bits.

NOTE

Learning More about WSL

While not required, the Windows Subsystem for Linux works very well with OpenWhisk. You can learn more about it in their blog. I strongly recommend it.

Your download will be a zip file containing just the executable file for your platform. Unzip the file and copy it into a folder that's part of your system path. In other words, a location where you can run the executable from your command prompt.

If you've done that correctly, you can go to your command prompt and type **`wsk`** to see it running correctly, as shown in Figure 2-9.

Figure 2-9. The CLI running correctly!

NOTE

Upgrading the CLI

Unfortunately, upgrading the CLI is currently a manual process. You can read about how to do this in a blog post.

The next step is a one-time security update. In order for the CLI to know who you are, how you log in to Bluemix, etc., you need to set an authentication token in the CLI itself. Figure 2-10 shows the command you need to type in step 2. You can either type it by hand, or click the Copy button and paste it into your terminal. Once that command is run, the `wsk` CLI will then be able to work with your account. You can confirm you're set up by running **`wsk property get`**.

Figure 2-10. Note that the authentication key above is obscured on purpose.

Finally, follow the last direction in Figure 2-8. Don't worry about understanding what it means. I'll be covering all of that later. Just ensure that you can run the command without error (as seen in Figure 2-11).

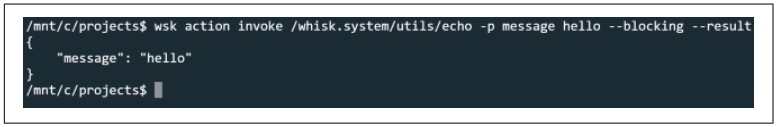


Figure 2-11. The test command running perfectly!

With this out of the way, you are now ready to start building your own serverless functions with Apache OpenWhisk!

CHAPTER 3

Working with Actions

In the first chapter, I referred to serverless as Function as a Service (FaaS). Within OpenWhisk, *actions* are how we refer to those functions. Everything you build will begin with actions, so understanding how to build and use them is fundamental to being a serverless developer on the OpenWhisk platform. I'll begin by covering the fundamentals of how actions should be built. You should think of these as guideliness, not hard and fast rules. I'll then switch to the actual rules you have to follow.

This chapter will focus on working with actions via the CLI. While this covers how developers interact with OpenWhisk, it certainly isn't how the public will use it. The next chapter will demonstrate how you can take these actions and make them available to the public.

The Fundamentals of OpenWhisk Actions

As previously described, what follows are the fundamentals (or principles) of how you should build your actions. These are guidelines you should follow but are not done at the code level. Consider them best practices, but note that you may bend these best practices from time to time.

First off, actions are typically small pieces of code. However, "small" is, of course, relative. It may mean 20 lines of code, or it may mean 200. But in general, if you find yourself working on a file that's get-

ting larger and larger, it may make sense to break it up into smaller actions.

Next, actions are atomic and stateless, meaning they run, do their stuff, and then disappear into the void. Nothing about them persists from one call to the other, so that's how you should code. But of course, there are caveats. When OpenWhisk runs your action, it doesn't kill it immediately. Rather, it waits a bit to see if it will be run again. Therefore, you actually *can* persist data and use it again. As long as your code doesn't *rely* on this cache and just uses it as optimization when available, then you will be good. The same applies to the filesystem. Your action can read and write to the filesystem but should assume that those files disappear when done. And as just stated, you can check to see if a cached file exists but don't require it. Finally, actions can work with persistance systems like databases. This means that while the action may go away, it can work with data that persists.

Finally, actions should focus on doing only one thing. Now, again, this is a guideline, not a rule. If your action does two things, the Serverless Police won't be stopping by your desk to slap your hand. But in general, the more focused your action, the more easily you'll be able to use it elsewhere. This is the same principle you learn when picking up any language. You use functions to save yourself from repeating code. In doing so, you naturally make your functions simpler and purpose driven. The same applies to actions.

Options for OpenWhisk Actions

When building OpenWhisk actions, you've got a few choices for how you develop them:

- JavaScript
- Swift
- Python
- Java
- PHP
- Docker

Each of these environments has its own specific details in terms of what version it's running and what's supported. For example, at the time this book was written, JavaScript actions used the Node 6.9.1 environment, and Python used Python 3.6.1. You can find out the

current settings for all environments via the OpenWhisk reference docs.

The final option, Docker, technically allows you to use any language to build your actions.

In this book, I'll be focusing on JavaScript. Everything demonstrated, though, will work just as well in any of the other environments. Do not worry if you aren't a JavaScript or Node expert. While JavaScript isn't something that can be precisely nailed down, anyone with at least a beginner's level understanding should be good to go. If you find yourself struggling with something that is more JavaScript and less OpenWhisk, open your browser to the Mozilla Developer Network. The Mozilla Developer Network (soon to be rebranded to MDN Web Docs) is the number one resource for *everything* web related. Don't let the "Mozilla" in the name fool you. Nothing about their docs is Firefox-specific, so developers should consider it a resource for any browser and any web technology.

Searching MDN Quicker

As a quick tip, if you need to search for something web related (perhaps how to work with arrays in JavaScript), prefix your search term with "mdn." So, for example, search "mdn array." The top result will be a link to the MDN.

Rules for JavaScript Actions

In general, you're free to write your JavaScript actions as you see fit, but there are a few rules you must follow. To help illustrate this, let's look at a simple action.

```
function main() {

    var message = "Hello World";

    return { result: message };

}
```

First off, the action has a function called "main." Your code must always have this. It doesn't mean you can't have more functions in the action named whatever you will, but OpenWhisk is going to run a function called "main" when it executes your action.

Secondly, you must return a JavaScript object. In the previous sample, the object has one key: `result`. That key name is totally arbitrary. The value, `message`, is a string. That's also arbitrary. All that really matters is that you return an object. To make this a bit more clear, here is another example.

```
function main() {

    var message = "Hello World";
    var highScores = [92332,89100,72910,51410];
    var bio = {
        name:"Raymond",
        cool:true
    };

    return {
        greeting:message,
        topScores:highScores,
        me:bio
    };

}
```

In this example, the object has three keys of various types of data. Again, the crucial point is returning an object. What's actually in that object is up to your action. If you're writing code to translate text, then your action will probably return a string. If you're writing code to target the Death Star, you will probably return an array of galactic coordinates.

And for the most part, that's it. There's going to be a few more rules to cover as we go along, but that's the basics. Now let's build a "real" action!

Your First Action

For our first action, let's start with something that is not too complex but is still a bit beyond the typical "Hello World" example. You can find this code in the GitHub repository as *ch03/action1.js*.

```
/*
Based on the time (of the OpenWhisk runtime at least!), return
an appropriate greeting.
*/

function main() {
    let message = '';
    let timeOfDay = new Date().getHours();

    if(timeOfDay < 12) {
        message = 'Good morning!';
    } else if(timeOfDay < 18) {
        message = 'Good Afternoon!';
    } else {
        message = 'Good evening!';
    }

    return { greeting:message };

}
```

This action begins by creating two variables, a message string and `timeOfDay`, a value based on the current time's number of hours. As the comment says on top, this will be based on where OpenWhisk is actually running it and may not match your local time.

Then, the code simply uses a few conditionals to create a greeting appropriate for the time.

NOTE

What the heck is "let"?

If the `let` keyword threw you there, don't worry. Yes, this is part of the latest JavaScript (really ECMAScript) standard, commonly referred to as ES6. The code could have just as easily used the `var` keyword instead. `let` is simply a bit more specific about its scope. If you're comfortable with ES6, go ahead and use it! This book will make liberal use of it. If you're not yet ready to embrace the new hotness, that's fine as well!

You can write this code yourself or use the file downloaded from the GitHub repository referred to in Chapter 1. In your terminal, ensure you are in the same directory as the file.

In order for this code to be deployed to OpenWhisk, you have to tell the CLI to deploy it. This can be done with the following command:

```
wsk action create nameOfAction nameOfFile
```

The first two arguments, `action create`, should be self-explanatory. This will create a new action. The third argument, `nameOfAction`, refers to the name you want to use for your action. The final argument is simply the filename. The CLI has *great* documentation within it. Figure 3-1 has an example of the result of running `wsk action`.

```
/mnt/c/projects/developing-serverless-applications-code/ch03$ wsk action
work with actions
Usage:
  wsk action [command]

Available Commands:
  create      create a new action
  update      update an existing action, or create an action if it does not exist
  invoke      invoke action
  get         get action
  delete      delete action
  list        list all actions

Flags:
  -h, --help   help for action

Global Flags:
      --apihost HOST         whisk API HOST
      --apiversion VERSION   whisk API VERSION
  -u, --auth KEY             authorization KEY
  -d, --debug                debug level output
  -i, --insecure             bypass certificate checking
  -v, --verbose              verbose output

Use "wsk action [command] --help" for more information about a command.
```

Figure 3-1. CLI Help FTW!

To deploy the action, run the following command:

```
wsk action create timedGreeting action1.js
```

Again, this assumes you're in the same directory as the code. If not, either add the full path in the last argument or simply change directories. The CLI will then tell you that the action is created.

Cool. So what do you do when you realize you made a typo or had a bug and need to update it? Just change the command from `create` to `update`.

```
wsk action update timedGreeting action1.js
```

You can run `update` even when the action doesn't exist, so if you want, you can skip using `create` and just stick to `update`.

In case you're curious, you can also delete actions: `wsk action delete timedGreeting` will remove it from the system. Want to see all your actions? Just run `wsk action list`, as seen in Figure 3-2.

Figure 3-2. Listing Actions

At first, this list will be short, but overtime it will grow. The list is currently sorted by the last time the action was updated. To get an alpha-sorted list, you would use `wsk action list --name-sort`. Also note that there is a default maximum of 30 actions returned per request. You can use the `--limit` argument to ask for more. As always, use the `-h` flag for help with all the various options you can use.

Running Your Action

Ok, the action previously created is deployed to OpenWhisk, but how do you actually test it? To be clear, we're talking about testing it as a developer. Using the action in actual production code will be covered in the next chapter.

To run an action, use the `invoke` argument. The general form of the command looks like so:

```
wsk action invoke name
```

So given that, you can run the action just created by running:

```
wsk action invoke timedGreeting
```

Doing so returns the following result:

```
ok: invoked /_/timedGreeting with id a773826ffa6841ee96e9...b
```

Not very clear, is it? When you invoke such an action, a few things happen. First, the CLI isn't going to wait around for it to finish. Though it may be quick and the code used for the first action may be incredibly simple, but the CLI isn't waiting.

Instead, the CLI returned what OpenWhisk refers to as an *activation*. Activations are records of your action's lifecycle. They contain the result of your action as well as a set of metadata about how it ran. To get the activation, you can use this command:

```
wsk activation get X
```

Where `X` is the ID returned from the `invoke` call. Here is what that looks like:

```
{
    "namespace": "ray@camdenfamily.com_dev",
    "name": "timedGreeting",
    "version": "0.0.2",
    "subject": "ray@camdenfamily.com",
    "activationId": "c8e7f96ecc9b4f81879427671937a4d6",
    "start": 1498847586291,
    "end": 1498847586294,
    "duration": 3,
    "response": {
        "status": "success",
        "statusCode": 0,
        "success": true,
        "result": {
            "greeting": "Good evening!"
        }
    },
    "logs": [],
    "annotations": [
        {
            "key": "limits",
            "value": {
                "logs": 10,
                "memory": 256,
                "timeout": 60000
            }
        },
        {
            "key": "path",
            "value": "ray@camdenfamily.com_dev/timedGreeting"
        }
    ],
    "publish": false
}
```


Activations are returned as a JSON packet. They contain a lot of data, much of which is obvious, but let's discuss some important ones.

- The most important part of the activation is the `response` key, specifically the `response.result`. When you invoke such an action, a few things happen.
- The next most important bits are `start`, `end`, and `duration`. However, you most likely only care about `duration`. This reflects how long it took for your function to run. OpenWhisk charges based on execution time rounded up to the nearest 100 ms. This function call took 3 ms but will be billed at 100 ms. But

again, OpenWhisk has a *huge* free tier, so this execution probably cost around 0.000000001 cents. Probably even less.

- The next one you should note is `logs`. If your code made use of `console.log()`, the debugging tool of champions, then they would show up there. This is also where you would see errors. We'll demonstrate an example of that in a bit.

If you're curious, you can actually get a list of all your activations using `wsk activation list`, as shown in Figure 3-3.

```
/mnt/c/projects/developing-serverless-applications-code/ch03$ wsk activation list
activations
c8e7f96ecc9b4f81879427671937a4d6 timedGreeting
c0e48e4404f14ff091c66c0c46c6066d timedGreeting
4f51fbd127a44ab88a2508c07986a391 timedGreeting
ccf7dd493a1f46678733f62ba4ff3889 timedGreeting
/mnt/c/projects/developing-serverless-applications-code/ch03$
```

Figure 3-3. A list of recent activations.

As with the action list command, the activation list is sorted by the most recent activation and limited to 30 results. Again though, this can be modified with a few additional arguments.

Still, the most convenient use of the `activation` command is the `poll`. The `poll` command simply waits for you to run actions and then reports the result automatically. You can open up a new terminal window and simply run that command and wait for results. It will not show any of the metadata or any of the results, but it will report errors and show any logs. Now would be a perfect time to show an example of that in action.

Let's begin by looking at a new action, one terribly flawed and doomed to failure.

```
/*
I have no chance of running well.
*/

function main() {
    console.log('running my action');
    let x = 1;
    let y = 2;
    let z = a+b;

    return { result: z};
}
```

There's two things to note here. First, the use of `console.log` as a simple debugging measure and secondly, the error that will occur when the function tries to add two variables that don't exist. You can find this code in *ch3/action_bad.js*. Use the `wsk` CLI to create a new action with the name `action_bad` and then invoke it. Figure 3-4 shows how `wsk activation poll` reports it.

Figure 3-4. wsk poll report

Note that the output includes both the log message as well as the error. When developing with OpenWhisk, you may want to keep a terminal open with the poll just to make debugging easier. But assuming your code works well, there's an even easier way to get results.

When invoking an OpenWhisk action, you can pass two useful arguments:

- `--blocking` (or `-b`) tells the CLI to wait for the function to complete. This will then return the entire activation to your screen.
- `--result` (or `-r`) tells the CLI to print just the result. If you don't care about execution times or the logs, this is definitely desirable.

Put together, you can then run your action like so:

```
wsk action invoke timedGreeting -b -r
```

And the result is much easier to work with:

```
{
    "greeting": "Good evening!"
}
```

This is even useful in case of errors. Here is the result of running `action_bad`:

```
{
    "error": "An error has occurred: ReferenceError: a
      is not defined"
}
```

Working with Arguments

So far our only action (the only working one) was simple. It accepted no input and simply returned a value based on time. Most actions, though, will need to accept arguments to function correctly. OpenWhisk makes this fairly easy. To pass arguments to an action, you simply use the `--param` argument. Here is an example:

```
wsk action invoke myAction --param something foo -b -r
```

In this example, the parameter `something` is passed with the value `foo`. Sending two parameters is as simple as repeating the argument:

```
wsk action invoke myAction --param something foo --param
somethingmore goo -b -r
```

Finally, you can pass a set of parameters based on a file. This could be useful if you have many parameters or more complex data, like arrays.

```
wsk action invoke myAction --param-file something.json
```

The file must be a JSON file in order for it to work. Now how does the code work with these parameters? All parameters, no matter how many are sent, are passed as one basic object. Here's a new action that shows this in action.

```
/*
Returns the area of rectangle.
*/

function main(args) {

    if(!args.width) {
        return {
            error:"Width argument required."
        }
    }

    if(!args.height) {
        return {
            error:"Height argument required."
```

```
        }
    }

    let area = args.width * args.height;

    return { area: area };
}
```

You can find this file in the GitHub repo as *ch3/area.js*. It's a simple action that determines the area of a rectangle. The first line now includes one argument, `args`, which will hold any and old arguments (or parameters) passed to the function. Notice that the code also checks for two particular parameters. This isn't required, but it makes the action a bit more proper. The code could be made even better if it checked for valid numbers too. Once it has those parameters, it uses a bit of math and returns the result. Here's an example of how the action can be invoked:

```
wsk action invoke area -b -r --param width 100 --param
height 90
```

And the result:

```
{
    "area": 9000
}
```

Actions also support the idea of default parameters. This allows you to set up parameters that can be changed on a case-by-case basis but revert to a default value when not passed. Default parameters are set using the `wsk` command line like so:

```
wsk action update actionName --param something somevalue
```

The previous command will update `actionName` so that it has a default parameter named `something` with a value `somevalue`. You can also use `--param-file` argument to specify a JSON file containing key/value pairs for default parameters. For this example, we'll revert to the boring but easy to understand "Hello World" example:

```
/*
Hello Word - because of course.
*/

function main(args) {

    let message = `Hello, ${args.name}!`;

    return { result:message };
}
```

You can find this file in *ch3/hello_world.js*. All it does is create a personalized greeting based on a name parameter. To create this action and set the default parameter for name, use this command:

```
wsk action create hello_word hello_world.js --param name
Nameless
```

Next, test calling it both with and without a name parameter. Here is the result without a parameter: `wsk action invoke hello_world -b -r`:

```
{
    "result": "Hello, Nameless!"
}
```

And here is the result of calling it with a parameter: `wsk action invoke hello_world -b -r --param name Ray`:

```
{
    "result": "Hello, Ray!"
}
```

You may be wondering why you would use a default parameter when code can be used to do the same thing. You can! Using a default parameter on the action itself allows you to change the implementation later. For example, from JavaScript to Java (but my goodness, why would you do that). Having it in code makes it a bit easier to see. It's really up to the developer to decide what makes sense.

Asynchronous Actions

So far, all of the actions demonstrated have returned a result immediately. What about actions that need to perform asynchronous actions? A perfect example of this is HTTP requests. Imagine an action that requests a third-party API, does some manipulation on the result, and returns it. This is actually a really good use case for serverless. A client application may have need of data that only a remote API can provide. But what if that API returns a lot of extra data that isn't necessary? Or perhaps it returns the data in XML instead of JSON. (Yes, some APIs still do that.) A serverless action could act as a proxy—both removing extra data not needed by the client and performing basic transformations on the data to make it easier to use by the client. Even better, if that third-party API ever goes out of business, the action could simply (hopefully!) switch to

another service and massage the data to return it exactly as it was before.

To use asynchronous actions with OpenWhisk, your code has to be slightly modified from previous examples. Instead of returning a simple JavaScript object, you must return a Promise. Promises are a (kinda) new way to handle asynchronous code. They work especially well in cases where you have to manage multiple asynchronous calls in a fixed order or workflow. Promises can be a bit scary at first, so if you've never used them before, be sure to read the helpful guide at MDN. However, their usage in OpenWhisk can be, thankfully, rather simple. Here is a sample pseudo-code action:

```
function main(args) {

    return new Promise(function(resolve, reject) {

        //stuff happens

        if(somethingGood) {
            resolve({result:1};
        } else {
            reject({error:'Oh no!'});
        }
    });

}
```

Let's tackle this bit by bit. The biggest change is that the code returns a Promise, not a plain JavaScript object. This basically tells OpenWhisk, "Just hang on, friend, I'm going to do some stuff and get back to you in a bit." Then, whatever code needs to run is executed. Finally, you decide if something good or bad happened. Notice how the Promise's argument, a callback, is passed a `resolve` and `reject` parameter. These are functions you run to basically say, "I ended well, and here's some data," or "something went wrong, let me share with you the details." In both cases, the result data is a plain JavaScript object.

For our first demo, let's write an action that returns the JSON Feed from my blog. A JSON Feed is basically a JSON version of a RSS feed. This action will get the feed but only return the most recent entry.

```
const request = require('request');

function main(args) {

    let url = 'https://www.raymondcamden.com/jsonfeed/
               index.json';

    return new Promise( (resolve, reject) => {

        request({url:url, json:true},
           (error, response, body) => {

            resolve({latestblog:body.items[0]});

        });

    });
}
```

The action starts off by loading the request library. OpenWhisk supports a number of popular npm modules out of the box. The reference guide mentioned earlier lists all of the npm modules that can be used. (OpenWhisk also supports custom npm modules, but that will not be covered in this book.)

The action then uses the request library to open the JSON feed for the blog, automatically parses the JSON, and then returns the first (i.e. most recent) blog entry. Here is a sample of the output:

```
{
  "latestblog": {
    "content_text": "(trimmed for size)",
    "date_published": "2017-06-29 09:03:00 -0700",
    "id":
    "https://www.raymondcamden.com/2017/06/29/
     handling-sms-with-openwhisk/",
    "tags": "openwhisk,watson",
    "title": "Handling SMS with OpenWhisk, IBM Watson,
              and Twilio",
    "url":
    "https://www.raymondcamden.com/2017/06/29/
     handling-sms-with-openwhisk/"
    }
}
```

Async actions are called just like any other OpenWhisk action. So given that the action was called "async1," calling it would have been done like so:

```
wsk action invoke async1 -b -r
```

Now that you know how to create OpenWhisk actions and call them via the CLI, in the next chapter you'll learn how to expose these actions so that others can use them as well.

CHAPTER 4

Using OpenWhisk Actions

So far, every example demonstrated has made use of the CLI to work with actions. We've covered creating, updating, and running actions via the CLI. But what about actually using OpenWhisk actions in production? There are three ways of doing that, each with their own benefits and recommended use cases.

Authenticated REST API

The first method is the authenticated REST API. Actually, this is what the CLI was doing on your behalf. The REST API allows for full integration with OpenWhisk, doing everything the CLI does and more. However, as you can probably guess by the title of this section, it does require authentication credentials.

When making requests to the REST API, your username and password must be passed. However, this is not the same as your IBM Bluemix login. In Chapter 2, you copied a command to set up your authentication information for the CLI. You can retieve that information using the CLI as well. Running `wsk property get` will return a set of values related to your current setup, including authentication information in the `whisk auth` value:

```
whisk auth            lotsofrandomnumbers:evenmorerandomnumber
whisk API host        openwhisk.ng.bluemix.net
whisk API version     v1
whisk namespace       _
whisk CLI version     2017-07-12T20:09:28+00:00
whisk API build        2017-07-31T14:35:32Z
whisk API build number  whisk-build-5338
```

While the values in the above sample were modified, note the colon that appears inside the `auth` value. When using the authenticated API, your username would be on the left of this and your password would be on the right. With these values, you could then make calls to the API to perform whatever actions make sense. So why would you use this?

While serverless is a very exciting and powerful new way to develop, there are a huge amount of app servers out there that will not be disappearing overnight. (And of course, serverless absolutely does not make sense in every situation!) It is entirely possible that your organization could have an existing app server using Node, PHP, or even ColdFusion and wants to make use of functionality deployed on OpenWhisk. By making simple authenticated HTTP calls to the REST API, you can then make use of those functions.

For developers wanting to use Node with OpenWhisk, take a look at the OpenWhisk npm package. This makes using the REST API far easier and supports a nice Promise-based method of working with every aspect of the platform. Here is a simple example of calling an action.

```
const openwhisk = require('openwhisk');
const options = {
    apihost:'openwhisk.ng.bluemix.net',
    api_key:'super_secret_key_no_one_will_guess'
}
const ow = openwhisk(options);

ow.actions.invoke({name:'getcats',blocking:true,result:true})
.then((cats) => {
  console.log('Here are the cats:', cats);
});
```

In this case, the `api_key` value would simply be the entire string of the `auth` value from the earlier CLI example. For more information about the npm package, see the docs on GitHub.

In general, the REST API will only be used for internal projects or projects involving existing app servers where authentication information can be safely embedded and used securely. The REST API should not be used in client-side applications.

Web Actions

Web Actions broadly provide the following features:

- A public, anonymous URL for invoking the action.
- A way to return more than just plain JSON data. Web Actions can return headers, headers and data, and non-JSON data, such as HTML and binary data. The ability to return headers will be crucial for client-side web applications.
- Actions invoked as web actions also get access to request information about how they were invoked. This includes request headers, CGI query string and path information, and the raw body of the request.
- A way to modify their behavior depending on how they are called. For example, by manipulating the URL, a user can ask for a portion of the data returned instead of the entire set.
- Finally, web actions with default parameters are considered *protected*, which means that they are still considered a parameter but can't be overridden by the user calling the action. Essentially, they are "read-only" parameters that can't be changed.

Enabling *web action* support for an existing action is simple. Let's begin by looking at a simple action.

```
const cats = ["Luna","Cracker","Robin","Pig","Simba"];

function main(args) {

    return {
        cats:cats
    };

}
```

This action returns a hard-coded list of cats. There are no parameters or any other activity; it simply returns the list. Set this up as an action called `cats`. (You can find this code in *ch4/cats.js*.)

```
wsk action create cats cats.js
```

Then ensure it works correctly:

```
wsk action invoke cats
```

Now that the action exists, let's discuss how to enable it as a web action and how to call it.

Enabling Web Actions

Enabling an action to be a web action involves adding the `web` annotation to it. Here's how to enable it:

```
wsk action update cats --web true
```

To confirm, use the CLI to retrieve the action:

```
wsk action get cats
```

The result should look like so:


```
{
    "namespace": "ray@camdenfamily.com_dev",
    "name": "cats",
    "version": "0.0.2",
    "exec": {
        "kind": "nodejs:6",
        "code": "\r\nconst cats = [\"Luna\",\"Cracker\",\
                "Robin\",\"Pig\",\"Simba\"];
                \r\n\r\nfunction main(args)
                {\r\n\r\n    return {\r\n        cats:cats\r\n
    };\r\n    \r\n}"
    },
    "annotations": [
        {
            "key": "web-export",
            "value": true
        },
        {
            "key": "raw-http",
            "value": false
        },
        {
            "key": "final",
            "value": true
        },
        {
            "key": "exec",
            "value": "nodejs:6"
        }
    ],
    "limits": {
        "timeout": 60000,
        "memory": 256,
        "logs": 10
    },
    "publish": false
}
```

You'll notice multiple bits of information about the action, including the source code itself. The important part, though, is `"web-export"` under the `"annotations"` key. If, for some reason, you want to disable web action support, you can simply run this CLI call:

```
wsk action update cats --web false
```

Accessing Web Actions

Once an action has been web enabled, how do you access it? The general form of a web action URL is:

```
https://{APIHOST}/api/v1/web/{QUALIFIED ACTION NAME}.{EXT}
```

Let's break this down bit by bit. The `APIHOST` value will be `open whisk.ng.bluemix.net` if you are using Bluemix to host your OpenWhisk code. `QUALIFIED ACTION NAME` is a bit more complex. This is a combination of three different values.

The first value is your Bluemix space. That was set up earlier, but you may have forgotten it. A quick way to see it again is to simply list your actions with `wsk action list`, as shown in Figure 4-1.

```
actions
/ray@camdenfamily.com_dev/cats                                  private nodejs:6
/ray@camdenfamily.com_dev/async1                                private nodejs:6
/ray@camdenfamily.com_dev/hello_world                           private nodejs:6
/ray@camdenfamily.com_dev/area                                  private nodejs:6
/ray@camdenfamily.com_dev/action_bad                            private nodejs:6
/ray@camdenfamily.com_dev/timedGreeting                         private nodejs:6
/mnt/c/projects/developing-serverless-applications-code/ch04$
```

Figure 4-1. Getting the action list

In Figure 4-1, notice how each action is prefixed with `/ray@camdenfamily.com_dev/`. This value will be different for you but is the space value you'll need to get your web action URL.

The next part of the qualified name is the package. Packages will be discussed in the next chapter, but for now, just know that actions exist in a default package.

Finally, the last part of the qualified name is just the action itself. That seems like a lot, but keep in mind that only the end of the URL typically changes as you work with different web actions.

Even better, the CLI supports a quick way to get the base URL for a web-enabled action. Simply run `wsk action get nameOfAction`

`--url`. The `--url` flag at the end simply asks for the URL used for the action. You won't get the rest of the action metadata.

Here's the URL for the `cats` action:

```
http://openwhisk.ng.bluemix.net/api/v1/web/ray@camdenfam
ily.com_dev/default/cats.json
```

Note the extension at the end tells the web action that it should return it's data in JSON. If you open this in your browser (although you should change it to the URL for your space), you should see a result similar to Figure 4-2.

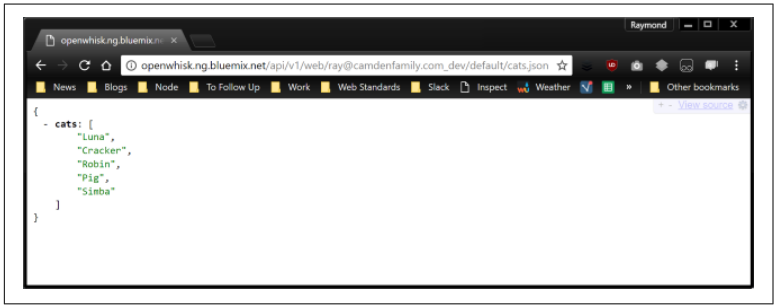


Figure 4-2. Testing the web action in a browser

In theory, this code is done. It has an anonymous accessible URL and could be used by client-side applications. To illustrate this, consider the following simple client-side application. (You can find this code in *ch4/cats.html*.)

```
<!DOCTYPE html>
<html>
  <head>
  <meta charset="utf-8">
  <title></title>
  <meta name="description" content="">
  <meta name="viewport" content="width=device-width">
  </head>

  <body>

  <h2>Cats</h2>

  <div id="result"></div>

  <script src="https://code.jquery.com/jquery-3.1.1.min.js"
  integrity="sha256-hVVnYaiADRTO2PzUGmuLJr8BLUSjGIZsDY...8="
  crossorigin="anonymous"></script>
```

```
    <script>
    // Change this to YOUR url...
    let apiUrl = 'http://openwhisk.ng.bluemix.net/api/v1/web'+
    '/ray@camdenfamily.com_dev/default/cats.json';
    $(document).ready(function() {
      console.log('Load cats');
      $.get(apiUrl).then(function(res) {
        console.log(res);
        let list = '<ul>';
        res.cats.forEach((cat) => {
          list += `<li>${cat}</li>`;
        });
        list += '</ul>';
        $('#result').html(list);
      });
    });
    </script>
    </body>
  </html>
```

This one page application (which is built this way just for simplicity's sake!) makes use of some simple jQuery calls to fetch the URL that points to the action defined earlier. Again, be sure to change the URL to match your own. After the JSON data is loaded, it's simply rendered out as an unordered list (see Figure 4-3).

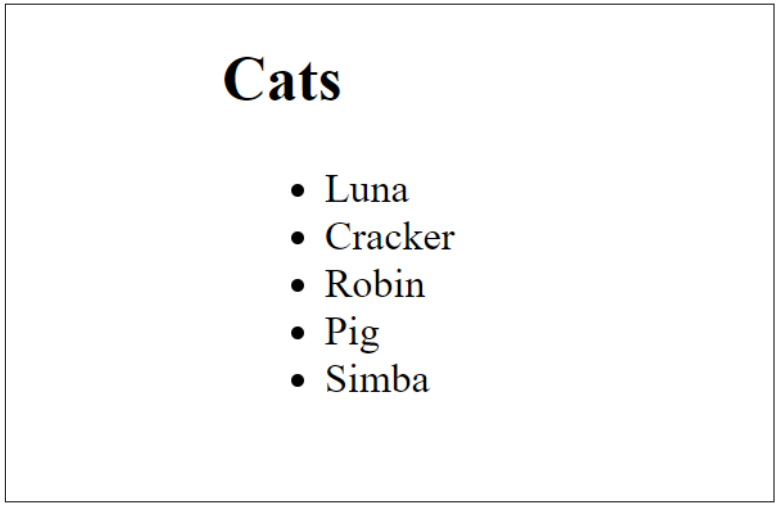


Figure 4-3. Client-side application

If you open Dev Tools in your browser, you'll notice that a CORS header was added automatically to the API. CORS stands for *cross-origin resource sharing* and is used to help define what resources are

allowed to use an API. You can learn more about CORS at the MDN but for now, just know that by default OpenWhisk is creating an API that can be used by JavaScript code on any host. This is useful, but you can disable this via annotations if you would like more precise control.

API Management

The most powerful way to expose your actions is with the API Management feature. The full name of this feature, Bluemix Native API Management, refers to the ability to not only say that a particular action should be addressable via a HTTP method, but also that you need to exert control over the API. For example, you may require a key to use the API. This also allows for things like rate limiting to ensure developers don't abuse your API (and drive up your costs). Finally, with API management you get a basic analytics system that tells you how many times your API has been called.

Enabling API Management

In the beginning of this book, I mentioned that OpenWhisk, when being used with IBM Bluemix, had a UI that mirrored what could be done with the CLI. I've focused mainly on the CLI aspects as that is most likely going to be more familiar for developers. In this section, I'm going to turn our focus to the UI as it makes working with the feature easier. While some of what I'm going to show can be done with the CLI too, it makes more sense for this section to focus on the visual tools for the API manager.

If you didn't bookmark it, you can find the Bluemix OpenWhisk console here: *https://console.bluemix.net/openwhisk/* (see Figure 4-4).

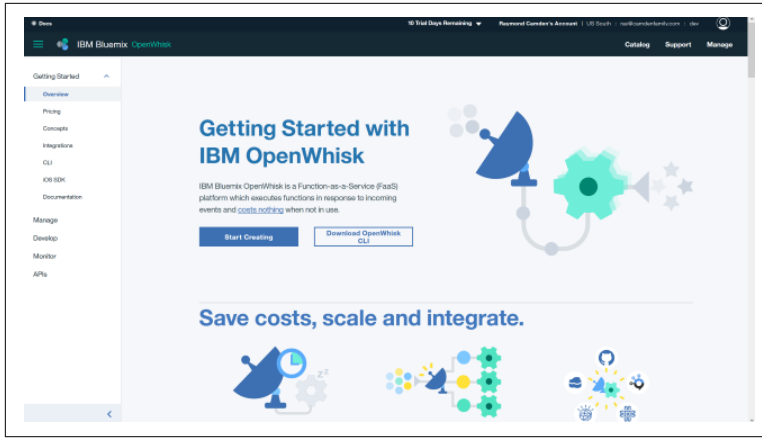


Figure 4-4. Bluemix OpenWhisk console

On the lefthand side, click the "APIs" link. Until you start actually creating APIs via this tool, you'll be presented with a basic page (see Figure 4-5) welcoming you to the feature.

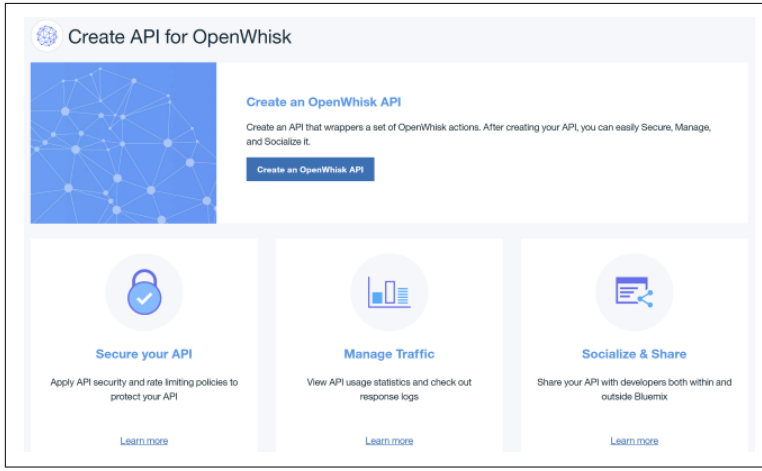


Figure 4-5. API management welcome page

To begin trying the feature, simply click the Create an OpenWhisk API button. The next page (Figure 4-6) is divided into two sections, "API Info" and "Security and Rate Limiting." Don't worry about the security stuff yet, I will come back to this later in the chapter and modify settings there. For now, focus on the basic API information.

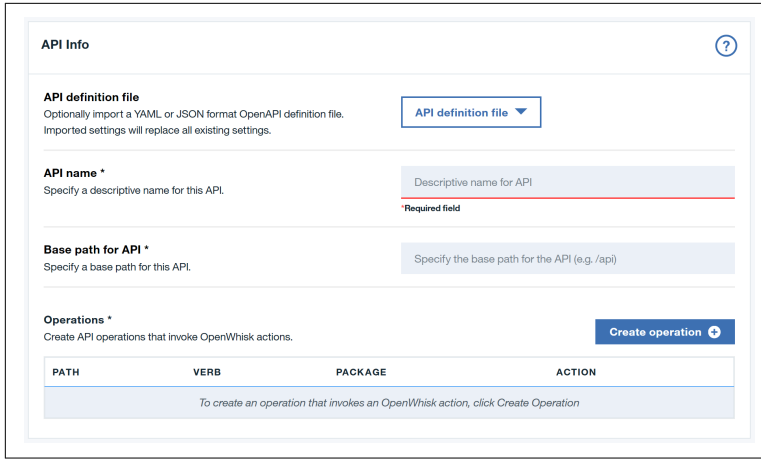


Figure 4-6. Setting up API information

The first item simply lets you import API information in OpenAPI format. The first thing you'll want to specify is the API name. An API can be comprised of multiple end points, so for example, a Cat API may have the ability to create cats, delete cats, search, and so forth. We built a simple Cat action earlier in the book that returned a list of cats. If we pretend that eventually we'll have many more such actions, we could imagine having a proper "Cat" API. So for now, we'll use "The Awesome Cat API."

The "Base Path" setting serves as a bucket for all the different parts of your API. You can use anything here, but it makes sense to give this a name that reflects the purpose of the actions you will expose under it. For our demo, we'll use "/cat".

So far, we've named the API and given it a base path, but it doesn't actually have any actions you can use with it. The UI refers to this as *operations*. This would be the "`list cats`," "`delete cats`," and so on, described earlier. To add the first operation, simply click the Create operation button.

The popup launched by the UI gives you a few options (see Figure 4-7):

Path
: This is the final part of the URL for your API and should reflect what you're doing with this particular operation. Since we know

we have an action that returns a list of cats, you can use "/list" here.

Verb
: This drop down lets you specify which HTTP verb should be used. There are rules about when you should use what. For example, deleting a cat should require the `DELETE` HTTP operation. You are not required to follow these rules though. (Although you should strongly consider following the norm.) For our case, keep it at the default of `GET`.

Package containing action
: This drop down lets you select a package. Leave that alone for now as it will be covered in Chapter 6.

Action
: This drop down lets you select what action to associate with the operation. The `cats` action is what will return a list of cats.

Response content type
: Finally, you can specify a particular content type for the response. The default is fine for this.

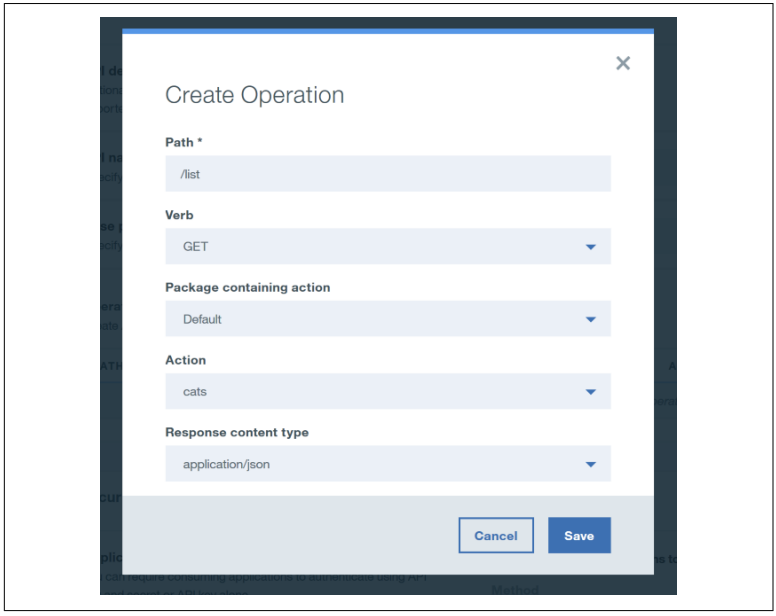


Figure 4-7. Details for the new operation

After hitting the Save button, you'll be brought back to the API info screen (Figure 4-8). Now your operation should be visible in the list.

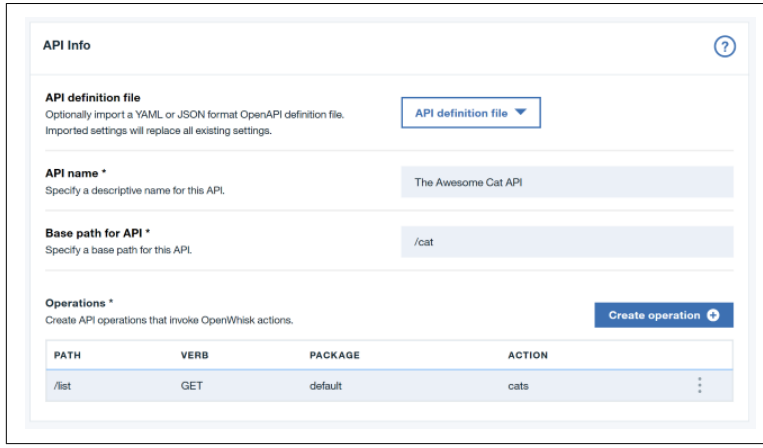


Figure 4-8. The completed information for the API

At this point, you are almost done. Scroll down past the security stuff and ensure you click the Save button. You'll now be taken to the summary page (Figure 4-9). This page includes the base route for your API, basic analytics, and even a response log of your most recent calls.

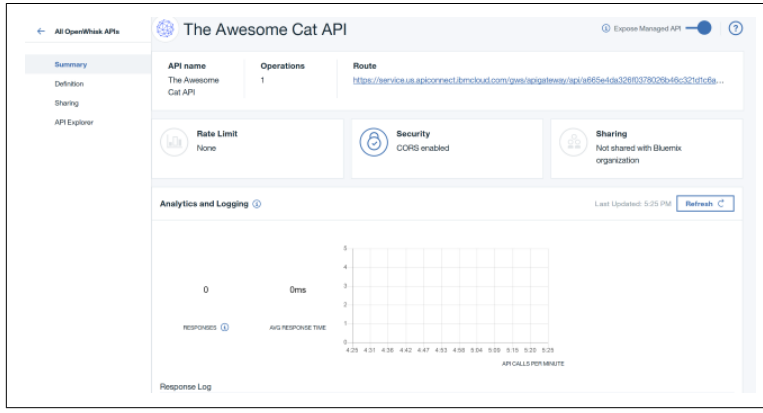


Figure 4-9. Your new API!

Testing APIs

The first thing to note on the API summary screen is the Route URL. It will look something like this:

```
https://service.us.apiconnect.ibmcloud.com/gws/apigateway/api/very long random string/cat
```

If you click on this link, you'll end up on a page with a 404 error. This is expected as the route URL is just the *base* of your API. Remember the operation defined earlier? Add `/list` to the end, and you should see your API result (Figure 4-10).

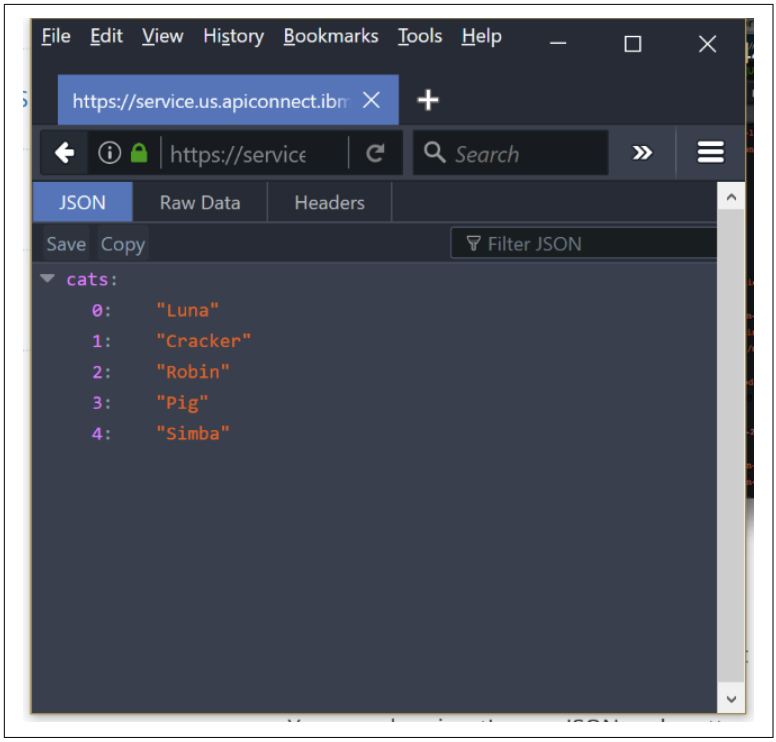


Figure 4-10. The browser's view of the data

If you reload your API multiple times, you can then return to the dashboard, use the Refresh button (Figure 4-11) by the analytics, and see how API calls are recorded.

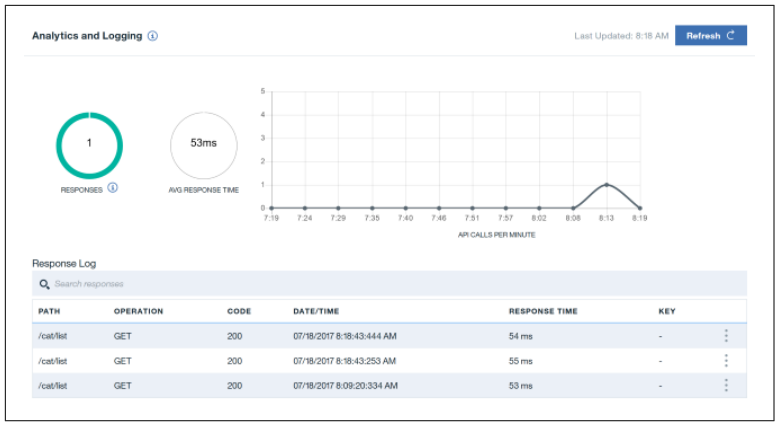


Figure 4-11. Analytics and logging for your API

The action used for the list operation, `cats`, is relatively trivial. Let's add a quick enhancement that lets users filter the list of cats. You can find the following code in source control at *ch4/cats2.js*.

```
const catList = ["Luna","Cracker","Robin","Pig","Simba"];

function main(args) {

    let cats = catList;

    if(args.filter && args.filter !== '') {
        cats = cats.filter((cat) => {
            return (cat.indexOf(args.filter) >= 0);
        });
    }

    return {
        cats:cats
    };

}
```

The first change was to create a copy of the list of cats (now renamed to `catList`). This lets us optionally modify the list. The next change is a simple check for a `filter` argument. If it exists and isn't blank, it is used to check against each cat, and only cats with a matching name are returned.

To update the action, simply use the CLI: `wsk action update cats cats2.js`. In case you're curious, it is completely fine to create an

action with one filename and then update it with another. The name of the action is important, not the file that acted as the source.

Now that the action is updated, the API created by the gateway can be used to work with this new feature. At the end of the URL, simply append: `?filter=Lu` (see Figure 4-12).

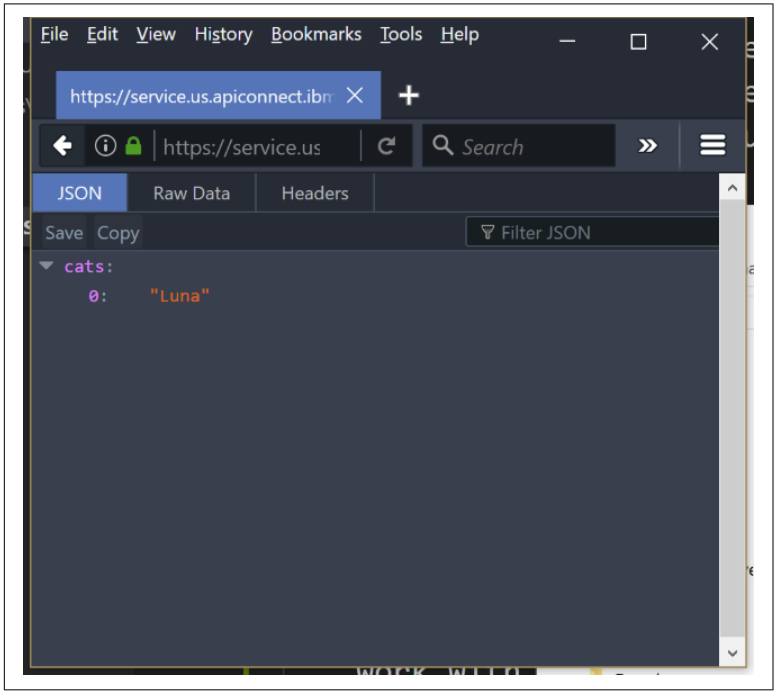


Figure 4-12. API result after filtering

For both web actions and APIs created with the API management feature, simple query parameters and form values are associated with arguments.

Locking Down APIs

Now that you've got a managed API, how can you go about limiting access to it? Bluemix Native API Management provides a quick and simple way of doing so. Ensure you're in the "Definition" page and toggle Require applications to authenticate via API Key, as shown in Figure 4-13.

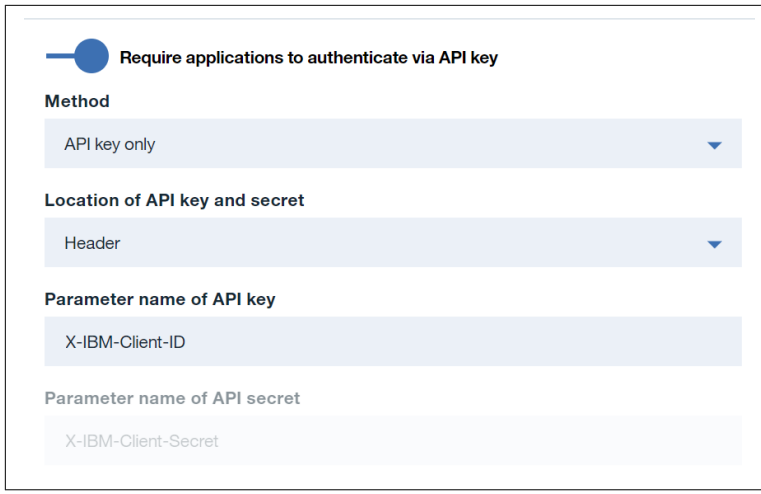


Figure 4-13. Enabling authentication

Once enabled, you can select how authentication is enforced. You can require either just an API key or an API key and secret value. The location of these values can only be in the header, not passed in as a query string or form value. Finally, you can tweak the names of the header values required for authentication. For now though, keep everything as the default.

Next, enable rate limiting. As you can guess, this lets you limit the number of times an API can be called within a time frame. Be sure to make note of the description of the `leaky bucket` method used to determine how often you can run the API. Imagine you've specified 100 calls per hour. The `leaky bucket` method does not let you call the API 99 times at once and then a final time later in the hour. Rather, the 100 calls must be spread out evenly across the entire hour (demonstrated in Figure 4-14).

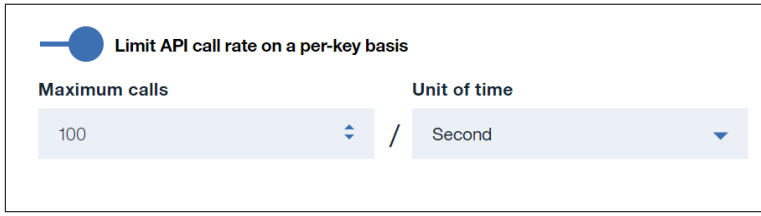


Figure 4-14. Enabling rate limiting

You can also enable authentication via OAuth (Figure 4-15). For now, only Google is supported, but leave that turned off for now.

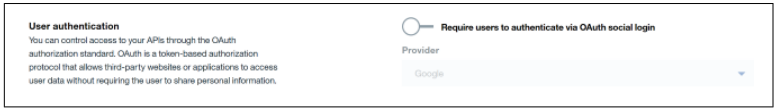


Figure 4-15. Google OAuth is also an option

Finally, be sure to click the Save button so your changes are applied. Now if you rerun the API call in your browser, you'll get an error, as shown in Figure 4-16.

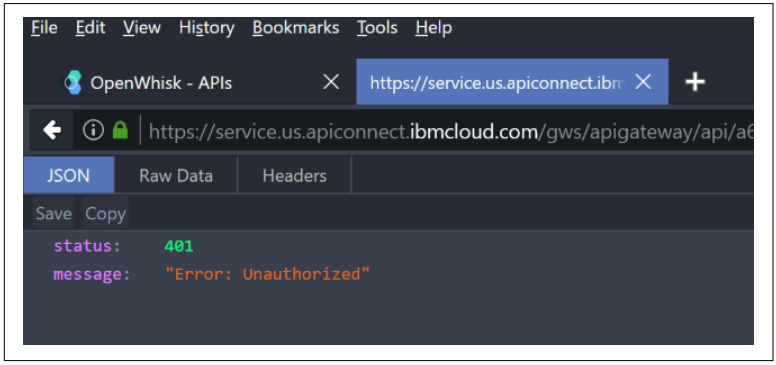


Figure 4-16. Your API is now protected

To start using this API, you'll need a key. Click on the "Sharing" link and you'll be provided with an interface that lets you work with keys in two contexts—either for users in Bluemix or those outside. In general, you'll probably be sharing the API with outside users, so begin by clicking the Create API Key in the second section. Figure 4-17 shows the dialog created when this is clicked.

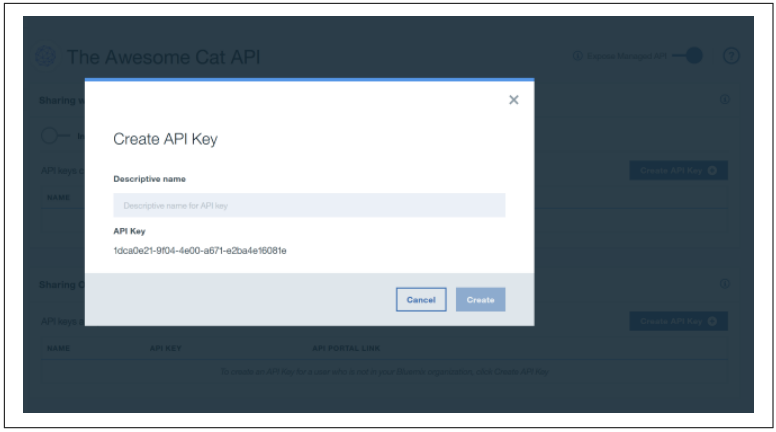


Figure 4-17. Adding a new key

The name is simply a description for who is using the key, so feel free to enter anything here, such as, **`Testing Key`**. After adding it, you'll see it listed back on the sharing page (Figure 4-18).

Figure 4-18. List of shared keys

Technically, at this point, you can simply open your command line and use a tool like Curl to call the API and include the proper authorization key. Another option is to use a desktop client such as Postman. However, note that next to the API key is an "API Portal Link." Click on this and you get a custom built developer portal (Figure 4-19) just for that key.

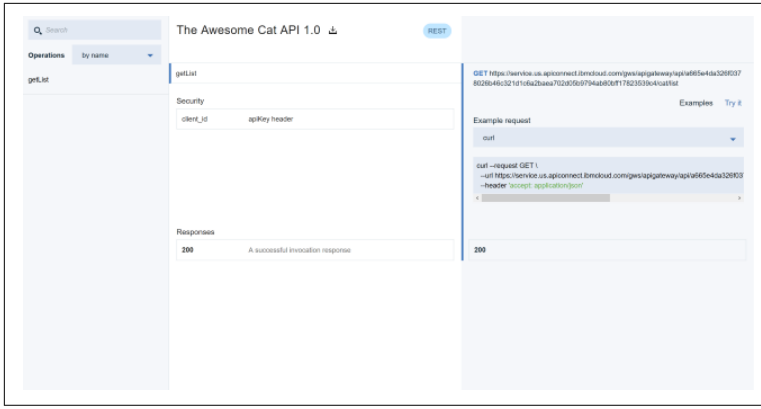


Figure 4-19. The developer portal

Also, make note of the "Try It" link on the righthand side. Click that, paste in the key, and then hit the Call operation button. You'll get a full report on both the request and response as well as the actual data of the result, as shown in Figure 4-20.

Figure 4-20. Testing the API

Remember that this API had only one operation. A more real world example would have multiple operations and opportunies to test in the portal.

CHAPTER 5
Building Sequences

You've just learned how to build actions that can be deployed as serverless functions with Apache OpenWhisk. One of the more powerful ways you can use an action is as part of a sequence. In essense, a sequence is simply an action that consists of other actions. Let's consider a simple example.

Imagine an action that simply takes a number and adds one to it. Let's call it the Incrementor.

Now imagine another action that takes a number and doubles it. We'll call that the Doubler.

If you wanted to both increment and double a number, you could simply pass your input to Incrementor, get the result, and then pass that result to Doubler. That would require two network calls (see Figure 5-1) but probably wouldn't be terribly slow.

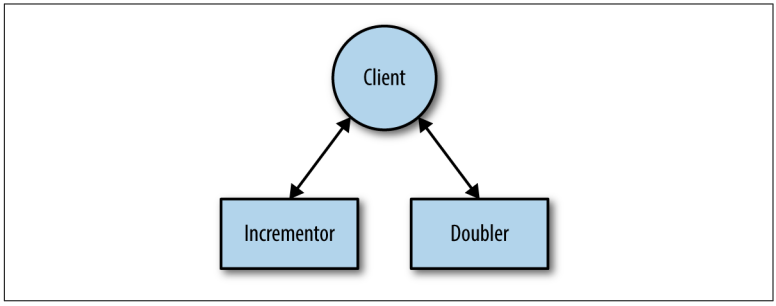


Figure 5-1. Working with two actions

But OpenWhisk provides a better way. By combining both actions into a sequence (as shown in Figure 5-2), the person calling the serverless function can work with both actions at once. To the client, it's simply another action, but behind the scenes, multiple actions are working together to create a result.

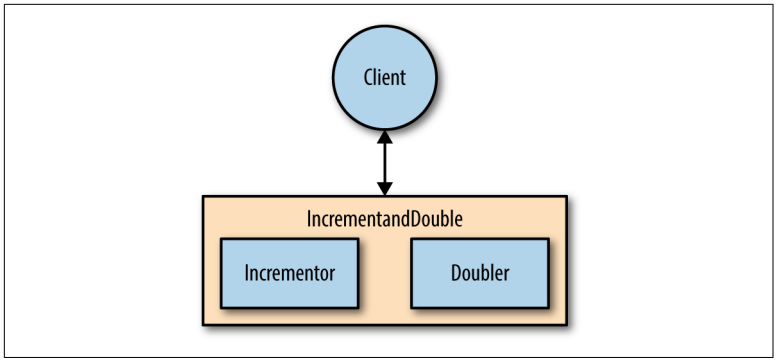


Figure 5-2. Working with one sequence

When we first introduced actions, we mentioned that you should make them small and focus them on one thing only. This becomes incredibly important when working with sequences. The more "single purpose" your actions are, the easier it is to combine them into sequences. Let's consider a more real-world example.

One of the services IBM provides under the Watson umbrella is a Tone Analysis service. This takes text input and attempts to provide information about the tone being used in the text. It has a pretty simple API, and it's trivial to build an OpenWhisk action to work with it. (You can find the full source code for that action on my website.)

Given that you can use an OpenWhisk action to analyze the tone of text, sequences allow us to do some interesting things. Imagine a simple action that parses an RSS feed. (You can find one built already.) By combining an RSS Parsing action with analysis, you can create a sequence that tells you the emotional tone of a blog.

Now imagine another OpenWhisk action that works with Twitter. (And again, you can find that code already built: *https://github.com/cfjedimaster/twitter-openwhisk*.) You can combine that action with the tone analysis to create a sequence that reveals the tone of a person's Twitter feed.

What's truly fascinating about the previous two examples is that from a few actions, two very powerful services were built simply by putting them together in a sequence. Let's look at how that's done and how you use them.

Creating Sequences

Creating sequences is pretty simple in OpenWhisk, but before you can create a sequence, you must have the *component* actions that make it up first. In other words, a sequence made from action A and action B requires that A and B actually exist first. That's probably obvious but is often something that can be missed. Let's begin by creating the components for the simple "Increment and Double" concept previously described. First, the Increment action.

```
function main(args) {
    let result = Number(args.number) + 1;

    return { result:result };

}
```

This is a fairly simple action. It takes an input parameter (assumed to be called "number"), adds one to it, and returns it as a result value. This file can be found in the *ch05* folder in the GitHub repository for this book. The name is *incrementor.js*. To send it to OpenWhisk, run `wsk action create incrementor incrementor.js`. Ensure it is working by doing a quick test: `wsk action invoke incrementor --param number 10 -b -r`. The result should be:

```
{
    "result": 11
}
```

Now create the code for the doubler action. This can be found in the GitHub repo as *doubler.js*.

```
function main(args) {
    let result = Number(args.number) * 2;

    return { result:result };

}
```

Push this to OpenWhisk using `wsk action create doubler doubler.js` and test it as well with `wsk action invoke doubler --param number 10 -b -r`. The result will be:

```
{
    "result": 20
}
```

To create a sequence in OpenWhisk, the `wsk action` command is used again but in a slightly different format. The basic syntax for the sequence version is: `wsk action create nameOfSequence --sequence firstAction,secondAction`. Essentially, you pass a `--sequence` flag with a list of actions that comprise your sequence. You can also specify a default parameter for the sequence in the same way you would any other action. The end result is a sequence, but in reality, it's just another action. Invoking a sequence is done just like any other action: `wsk action invoke nameOfSequence`. Let's go ahead and do that for our previous actions.

```
wsk action create --sequence incrementAndDouble --sequence
incrementor,doubler
```

The result should simply be: `ok: created action incremen tAndDouble`. Now let's invoke it with `wsk action invoke incremen tAndDouble --param number 10 -b -r`. The result should be (10+1)*2=22:

```
{
    "result": null
}
```

Woah—what went wrong? One requirement of sequences is that output from the previous action must match the desired input of the second. So what does that mean? The input to incrementor was a parameter called `number`. The result was a variable called `result`. The input to doubler was also a parameter called `number`, but OpenWhisk sent in a parameter called `result` because that's what the previous action returned. Now would be a great time to share a great debug tip. Modify the code of *doubler.js* to add this line after the initial function declaration:

```
console.log(JSON.stringify(args));
```

All this code does is log out the arguments sent to the action. You can then update the action like you would normally: `wsk action update doubler doubler.js`. What's interesting to note is that you are not only updating doubler but the sequence as well. In another tab, start listening for activations using `wsk activation poll` and then invoke the sequence again. Figure 5-3 shows what you should see after running an invocation.

```
Activation: doubler (52a91c889b454a3681d7aa4a81a63693)
[
    "2017-07-25T20:29:29.672564478Z stdout: {\"result\":11}"
]
```

Figure 5-3. Debugging log from Doubler

As you can see in Figure 5-3, the input was the right number value, but the name of the parameter was `result`, and not `number`. So how do you fix this? You could simply edit *doubler.js* to look for the result parameter. But what happens to people using doubler by itself? It's going to seem odd to pass a parameter named `result`. You could make the code look for *either* argument and simply check one first. While that would work, it makes *doubler.js* tied to the sequence, and that's just a bad idea. You could imagine other sequences using doubler and the code getting messier and messier. There are two general fixes for cases like this.

The first solution is a bit complex, but you can add a third action that sits between two unrelated actions. It simply massages the input provided to it to make it appropriate for the next action. So for example, something like this:

```
function main(args) {
    //rename args.result to args.number
    return { number: args.result};
}
```

This works well and is especially useful in more complex sequences. A simpler solution though is to consider renaming the result. Instead of using a generic "result" result, perhaps return it as something more appropriate. Consider this new version of incrementor (found in *incrementor2.js*):

```
function main(args) {
    let result = Number(args.number) + 1;

    return { number:result };

}
```

This code has the exact same logic. The only thing that changed is the name of the returning value. For completeness sake, the same change was made in *doubler2.js*. After both actions are updated, the result is finally as expected.

```
{
    "number": 22
}
```

Overall, sequences allow for the most complex serverless applications as developers can grow an app by simply creating longer and longer sequences. They can also reuse the same actions in multiple different sequences and get the benefit when one action is updated with bug fixes or performance changes.

CHAPTER 6
Working with Packages

The next feature of OpenWhisk that I'll discuss is packages. At their heart, packages are merely a way to organize actions. Consider an action that reads tweets from a Twitter account. Now consider another action that searches all of Twitter for a keyword. Finally, imagine an action that can post a tweet for an account. Packages let us do powerful organizational-type tasks with those actions.

First, a package serves an organizational purpose. Any action created is automatically placed in a default package. Actions must also have a unique name. So if you wanted an action named "process," you couldn't have more than one. (To be fair, "process" is a pretty poor name by itself.) By using a package, you can place the action inside it, allowing for more than one action with the same name. So for example, an `orderSystem` package could have an action called "process," and a `userSystem` package could have one as well.

Second, packages allow for default parameters that apply to the entire package at once. The Twitter API requires certain authentication values before you can make use of it. If you had multiple actions in a Twitter package, you could supply a default key for all of them at once and not have to worry about setting it on each and every action.

Finally, packages can also be shared. This means that Twitter package described previously could actually be made available to everyone using OpenWhisk. Of course, that then makes it impossible to use a package-wide default parameter for the key because if I shared my package with my key, you would be able to use my credentials to

work with Twitter. Luckily, you can use another feature called package *binding*. Binding is simply the act of saying, "I want to use this public package but create my own version of it." I can then supply a default key to make my usage of it easier.

You'll also discover that Bluemix has a set of packages ready to be used with your code already. These packages cover useful utilities as well as wrappers for some of the popular Watson APIs. At the end of this chapter, I'll briefly cover them and point you to the documentation.

Creating and Managing Packages

Creating packages is fairly simple. The command is: `wsk package create name` where "`name`" is the name of the package. Like actions, the name of each package must be unique. While you can think of packages almost like subdirectories, note that you cannot create a package within a package. Go ahead and create a new package called `firstPackage`:

```
wsk package create firstPackage
```

The result should be a simple success message. You can then list your packages using: `wsk package list` (see Figure 6-1).

Figure 6-1. Listing packages using the wsk package list command

Right now the package is empty, so let's put a simple action in it just for testing purposes. The following action simply returns all the arguments sent to it. You can find it in the GitHub repository for this book as *ch06/echo.js*.

```
function main(args) {

    return { params: args};

}
```

Putting an action into a package is relatively simple. When creating or updating an action using `wsk action`, you simply prefix the name of the action with the name of the package and a slash. So to put this into the new package, the following command should be used:

```
wsk action create firstPackage/echo echo.js
```

After creating it, you can confirm that it's in the package a few ways. First, ask for all your actions with `wsk action list` (see Figure 6-2).

```
/mnt/c/projects/developing-serverless-applications-code/ch06$ wsk action list
actions
/ray@camdenfamily.com_dev/firstPackage/echo                    private nodejs:6
/ray@camdenfamily.com_dev/doubler                              private nodejs:6
/ray@camdenfamily.com_dev/incrementor                          private nodejs:6
/ray@camdenfamily.com_dev/incrementAndDouble                   private sequence
/ray@camdenfamily.com_dev/cats                                 private nodejs:6
/ray@camdenfamily.com_dev/async1                               private nodejs:6
/ray@camdenfamily.com_dev/hello_world                          private nodejs:6
/ray@camdenfamily.com_dev/area                                 private nodejs:6
/ray@camdenfamily.com_dev/action_bad                           private nodejs:6
/ray@camdenfamily.com_dev/timedGreeting                        private nodejs:6
/mnt/c/projects/developing-serverless-applications-code/ch06$
```

Figure 6-2. Action list with new package and action on top

Notice how the new action (the first one in the list) includes the package name. Another way to find your action is to get the package itself using `wsk package get firstPackage`, as shown in Figure 6-3.

```
/mnt/c/projects/developing-serverless-applications-code/ch06$ wsk package get firstPackage
ok: got package firstPackage
{
    "namespace": "ray@camdenfamily.com_dev",
    "name": "firstPackage",
    "version": "0.0.1",
    "publish": false,
    "binding": {},
    "actions": [
        {
            "name": "echo",
            "version": "0.0.1",
            "annotations": [
                {
                    "key": "exec",
                    "value": "nodejs:6"
                }
            ]
        }
    ]
}
```

Figure 6-3. Data about the package

The command line call returned metadata about the package as well as a list of every action inside it (currently just one). That data could get a bit verbose. For a shorter version, you can use `wsk package get firstPackage --summary`, as shown in Figure 6-4.

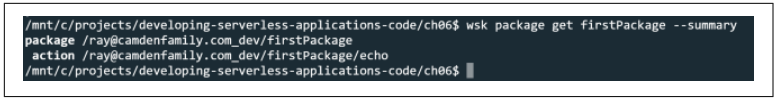


Figure 6-4. A shorter, nicer list of actions in the package.

Using an action in a package is also just as easy as putting the action inside it. As with creating (or updating), you simply prefix the name of the package when invoking the action. Here's an example that also illustrates the echo feature of the action just created.

```
wsk action invoke firstPackage/echo --param name Ray --param
awesome true -b -r
```

```
{
    "params": {
        "awesome": true,
        "name": "Ray"
    }
}
```

As I mentioned before, packages can have default parameters as well. Let's update the package to add that: `wsk package update firstPackage -param music trance`. This command specifies that every action in the package will have a default parameter named `"music"` with the value `"trance"`. In case you're curious, you can still use default parameters for each action, and they will take priority over the package-wide action. To see this new default in action, simply run the previous test on echo, and you'll see it returned:

```
{
    "params": {
        "awesome": true,
        "music": "trance",
        "name": "Ray"
    }
}
```

If you ever need to check what parameters are set for a package, use the `wsk package get` command *without* the summary flag. It will be returned in the JSON description of the packge:

```
{
    "namespace": "ray@camdenfamily.com_dev",
    "name": "firstPackage",
    "version": "0.0.2",
    "publish": false,
    "parameters": [
        {
```

```
                "key": "music",
                "value": "trance"
            }
        ],
        "binding": {},
        "actions": [
            {
                "name": "echo",
                "version": "0.0.1",
                "annotations": [
                    {
                        "key": "exec",
                        "value": "nodejs:6"
                    }
                ]
            }
        ]
    }
```

If you decide to share your package, you can update the package with the "shared" annotation. For example, this will share `firstPackage`:

```
wsk package update firstPackage --shared yes
```

This too can be seen when getting the package data but note that it will show up as a `"publish"` value, not `"shared"`. Unfortunately, there isn't an easy way for people to know your package is shared. For others to use it, you would need to specifically tell them how to address it. In that case, they can't use `firstPackage/echo` as that doesn't include your name or any other identifying information.

You may not have noticed, but when you run `wsk package list`, information about your account, or "namespace," was included in the result. For example, here is the full name of the `firstPackage` package: `/ray@camdenfamily.com_dev/firstPackage`. By using that prefix as well, another user can invoke your new shared package. Here is how another account would call the package:

```
wsk action invoke /ray@camdenfamily.com_dev/firstPackage/echo
--param x 1 -b -r
```

Figure 6-5 shows what you should see after running an invocation.

Figure 6-5. Invoking the shared package.

Basically, instead of just `package/action`, it's `namespace/package/action`. If you feel that is too verbose, you can use yet another feature of packages: *bindings*. A binding creates a reference to the package. (So in other words, if the original package is updated, your copy is as well.) However, you get the benefit of supplying your own default parameters as well as giving it a shorter name within your account. A great use for this is configuration values for APIs that can be shared across multiple actions.

To bind a package, use the following syntax:

```
wsk package bind thePackage newName
```

If you want to specify a default parameter, you must do it when you bind. There is no capability to update a binding. Setting those defaults works just like in any other command: `--param someParam someValue` as an example. Lastly, note that you can also bind a package to yourself. Why would you do that? Remember that package default parameters are shared. That means you can't share a cool package with others if you've specified confidential information as a parameter. Instead, you can bind a copy of your own package to yourself and specify the parameter then.

To see this in action, let's create a bound copy of firstPackage and specify a new default music style, one that is much more sensible:

```
wsk package bind /ray@camdenfamily.com_dev/firstPackage fp
--param music disco
```

In this example, I basically said, "Make a bound copy of firstPackage to a new package called fp. Also, set a default parameter music with the value of disco." To see this in action, you can invoke the echo command on the new package with: `wsk action invoke fp/echo -b -r`:

```
{
    "params": {
        "music": "disco"
```

```
    }
}
```

Using Bluemix Packages

In the very beginning of this book, I made it clear that working with IBM Bluemix was optional. There's quite a bit of cool and powerful features on Bluemix, but it is not something you *have* to use as an OpenWhisk developer. One reason you may want to consider it, however, is the powerful suite of shared packages Bluemix provides developers. Even better, Bluemix will even automatically deploy packages for you when you work with certain services. For example, if you make use of Cloudant, Bluemix can automatically give you a complete package of actions that works with it and configures it for you with your credentials.

All of Bluemix's packages will be in the "whisk.system" namespace. You can list them by using the command `wsk package list /whisk.system`. Full documentation for these packages can be found at the main Bluemix site, but here is a list of currently supported packages and a brief description of each.

Alarms
: Related to triggers, covered in Chapter 8.

Cloudant
: Used to work with the Cloudant NoSQL database.

Combinators
: Used to add some simple logic to actions (for example, doing something when an action fails to run correctly).

GitHub
: Lets you set up a webhook for responding to GitHub repository activity.

Messaging
: Used with IBM Message Hub.

Push notifications
: Related to Bluemix's push notification service for mobile devices.

Samples
: A few simple example actions.

Slack
: Because everything has to integrate with Slack.

Utils
: A set of utility actions, including an echo action like the one built previously.

Watson-translator, Watson-speechToText, Watson-textToSpeech
: All related to various Watson services working with speech and text.

Weather
: Integrates with the Weather Company APIs.

Websocket
: Works with a web socket server.

Invoking these Bluemix-hosted actions is just like calling your own package actions; for example: `wsk action invoke /whisk.system/utils/echo -b -r --param woot epic` will return:

```
{
    "woot": "epic"
}
```

Again, be sure to check the docs for more information on using these packages. While the command line can return some information (what actions and brief descriptions), you will want the online documentation for a deeper dive into the usage.

CHAPTER 7

Using Triggers and Rules

You've now seen how to create actions as well as how to call them via HTTP calls. But OpenWhisk also allows for an event-based activation system configured by *triggers*. This creates an entirely new way of working with serverless. You can write code that responds to new data being created, new files being stored, and more. With triggers and rules, your serverless code can be self-running and automated.

What are Triggers?

Triggers are simply events. For developers who have worked in languages with events, like JavaScript, this should be a familiar concept. Generally speaking, a developer simply says, "I want so and so to happen when this particular event occurs," and OpenWhisk supports this via triggers.

By itself, a trigger is simply an event and doesn't actually do anything. In the next section, you'll learn how to associate a trigger with a rule to fire calls to actions. You can begin working with triggers using the CLI, which follows a similar usage pattern to both actions and packages.

Creating a Trigger

To create a trigger, simply use `wsk trigger create NAME`:

```
wsk trigger create firstTrigger
```

After running this, the CLI will simply give you a message stating that the trigger was created. You can see what triggers you've created by doing `wsk trigger list`. You can remove triggers by simply doing `wsk trigger delete NAME`.

Firing a Trigger

To fire a trigger, use the following syntax `wsk trigger fire NAME`. Doing so returns an activation ID:

```
ok: triggered /_/firstTrigger with id 0526fd4aae5447df98...fc
```

And that's it—nothing more. By themselves, triggers are simply events. To have something actually happen when a trigger is fired, you have to combime them with a rule.

What are Rules?

Rules are the counterpart to triggers. They basically say, "When a trigger is fired, I want you to run an action." If triggers are events, rules set up the event listeners. A rule creates a one-to-one relationship between a trigger and an action. If you need to have one trigger fire off multiple events, then you simply create multiple rules, each associating the one trigger with a different rule.

Creating a Rule

To create a rule, the syntax is a bit different from that of a trigger. The basic form is `wsk rule create nameOfRule nameOfTrigger nameOfAction`. For example, to create a rule based off the earlier trigger, you could use the following:

```
wsk rule create firstRule firstTrigger timedGreeting
```

This rule says, when the `firstTrigger` is fired, run the action `timedGreeting`. If you forget this, you can fetch the details of a rule like so: `wsk rule get firstRule`. This returns a JSON packet describing the rule:

```
{
    "namespace": "ray@camdenfamily.com_dev",
    "name": "firstRule",
    "version": "0.0.1",
    "status": "active",
    "trigger": {
        "name": "firstTrigger",
```

```
            "path": "ray@camdenfamily.com_dev"
        },
        "action": {
            "name": "timedGreeting",
            "path": "ray@camdenfamily.com_dev"
        },
        "publish": false
    }
```

Pay special attention to the `status` field above. Rules have a special feature where they can be disabled. This allows you to keep an "event listener" for a trigger but temporarily disable it. To use this command, you would use `wsk rule disable firstRule`. Doing so and then fetching the rule details again would show a status of `inactive`. You could re-enable the rule by doing `wsk rule enable firstRule`.

Testing the Rule

Now that a rule is associated with the trigger, you can see it in action by simply firing the rule again. If you have a window open monitoring your activations with `wsk activation poll`, you can see this yourself. Run `wsk trigger fire firstTrigger`, and you'll see three activations:

```
Activation: timedGreeting (59e34d94169342f699af1ea540f4d128)
[]

Activation: firstTrigger (25b571c554524704a6fbfa0ebdb7ca11)
[]

Activation: firstRule (45846e884e434ea9b3a728975302474e)
[]
```

The order is asynchronous, so you may see a different order. But essentially what you are seeing here is the trigger firing, the rule noticing this because it has been told to listen to it, and then the action fired off because the rule was told to do so when activated.

Feeds and OpenWhisk Supplied Triggers

Feeds are simply a way of working with triggers in an automated manner. Earlier in the chapter you saw how triggers could be fired from the CLI, but for most cases, users will expect triggers to be fired automatically based on some external service. OpenWhisk supports this feature via feeds. Examples of feeds would be:

- Data being added to a Cloudant database
- A change being made to a GitHub repository
- A new tweet from a particular user
- Email arriving to a particular account

While creating a feed is outside the scope of this book (users can see the specifics in the OpenWhisk documentation), OpenWhisk on IBM Bluemix provides multiple feeds that can be used in triggers and associated with your own actions via rules. Currently these feeds are:

`/whisk.system/alarms/alarm`
: A simple CRON-based trigger. This is useful for creating a scheduled trigger.

`/whisk.system/cloudant/changes`
: Fired when changes are made to a Cloudant database.

`/whisk.system/github/webhook`
: Used to set up a webhook for use with a GitHub repository. Supports listening to multiple different types of GitHub events.

`/whisk.system/messaging/messageHubFeed`
: Used to listen for messages posted to a Message Hub instance.

`/whisk.system/pushnotifications/webhook`
: Used to listen to device subscription/unsubscription events.

Don't forget, though, that custom feeds can be created, so you are not limited by what OpenWhisk provides out of the box. For example, while there is a prebuilt feed for Cloudant changes, one could be built for Mongo as well.

CHAPTER 8

Going Further with OpenWhisk

Now that you've gotten a good introduction to Apache OpenWhisk, where can you go to learn more? Begin by bookmarking the two primary web sites for OpenWhisk development:

The Apache OpenWhisk open source project
: This is where you can learn how to participate in the project, file issues, and keep up to date with new changes.

OpenWhisk at IBM
: From here, you can register for a Bluemix account, begin testing OpenWhisk, and learn about other services that could work well with OpenWhisk.

Asking Questions

If you have questions about OpenWhisk, there's a few places you should consider:

- First is the OpenWhisk Slack. You can sign up for it through their channel. You'll find it a fairly active Slack with engineers on the product as well as end users.
- There is also a more generic Slack for serverless. You can sign up for this Slack channel as well.
- You can also use the OpenWhisk tag at StackOverflow. Right now, there isn't a lot of content there, but it is growing every day.

Additional Reading

For additional reading, consider these resources:

- While I've shared this before, the core docs can be found at *https://console.bluemix.net/docs/openwhisk/index.html#getting-started-with-openwhisk*.
- There is also a Medium publication focused on OpenWhisk with multiple authors.
- While not focused on OpenWhisk alone, IBM published a "developer journey" that provides an excellent real world walk through of building an application. Obviously, OpenWhisk is part of the process. Many blog entries tend to be simple and not necessarily similar to what you would see in the real world. The developer journey does a great job of presenting a realistic scenario.

Participating

Since OpenWhisk is an open source project, you may want to actually participate in making it better. Or perhaps you simply want to make note of a bug you ran into while using the product. Here are two links to get you started:

- First, consider joining the developer listserv. This is where people discuss OpenWhisk, including both problems and ideas for new features. You do not have to actually participate in conversations. You are free to join and listen in for an idea of what's coming next.
- Second, if you do find a bug, you should report it on the GitHub repo for OpenWhisk. Be sure to search the open issues first to ensure it hasn't already been reported.

Everything Else

Finally, you can also keep up to date with OpenWhisk news by simply following its Twitter account. The account tweets updates and retweets interesting news from other developers as well (Figure 8-1).

Figure 8-1. The OpenWhisk Twitter account

While OpenWhisk doesn't care what editor you use to write your code, one of the best (free and open source) editors out there is Visual Studio Code. There is an extension in development for Visual Studio Code that wraps some of the features of the CLI right in the editor itself. Figure 8-2 demonstrates it in action. You can find the extension on GitHub.

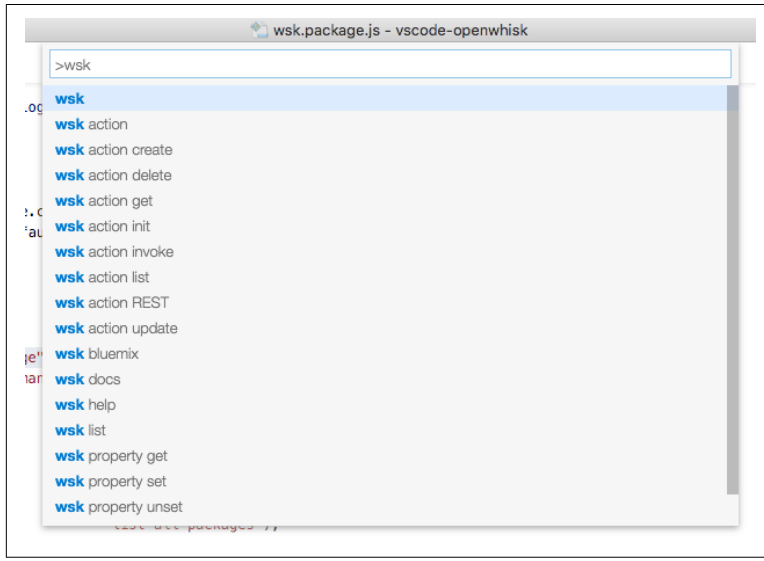


Figure 8-2. An example of the extension in action, taken from the project's readme.md file

In Figure 8-2, you can see the extension providing quick access to common CLI commands within the editor. This lets you do all of your work within the editor itself, bypassing your terminal if you would like.

Finally, consider signing up for Serverless Status. This is a weekly newsletter (curated by myself and others) that includes links to interesting news in the serverless world. While not constrained to just OpenWhisk articles, you'll find interesting content from across the entire spectrum of serverless development.

About the Author

Raymond Camden is a developer advocate for IBM. His work focuses on LoopBack, serverless, hybrid mobile development, Node.js, HTML5, and web standards in general. He's a published author and presents at conferences and user groups on a variety of topics. Raymond can be reached at his blog, @raymondcamden on Twitter, or via email at *raymondcamden@gmail.com*.