MÁSTER EN INGENIERÍA INFORMÁTICA

Cloud Computing: Servicios y Aplicaciones

Practica 02 :
Implementing facial recognition using Functions-as-a-Service

Autores

Hamada Bouhacida
177339003
bouhcidahamada@correo.ugr.es
hamadabouhcida34@gmail.com

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
----
Granada,Junio de 2022

Go Functions-as-a-Service :

Running a Go application on AWS Lambda is easier than ever, once you figure out how to configure Lambda, API Gateway and 10 or other "serverless" services to support the Go functions.

This is a boilerplate app with all the AWS pieces configured correctly and explained in depth. See the docs folder for detailed guides about functions, tracing, security, automation and more with AWS and Go.

With this foundation you can skip over all the undifferentiated setup, and focus entirely on your Go code.

Motivation :

Functions-as-a-Service (FaaS) like AWS Lambda are one of the latest advances in cloud Infrastructure-as-a-Service (IaaS). Go is particularly well-suited to run in Lambda due to its speed, size and cross-compiler. Check out the Intro to Go Functions-as-a-Service and Lambda doc for more explanation.

For a long time, Go in Lambda was only possible through hacks -- execution shims, 3rd party frameworks and middleware, and little dev/prod parity. But in January 2018, AWS launched official Go support for Lambda and Go released v1.10 paving the clearest path yet for us Gophers.

This project demonstrates a simple and clean foundation for Go in Lambda. You can clone and deploy it with a few commands to get a feel for the stack. Or you can fork and rework it to turn it into your own web app.

What's remarkable is how little work is required to get all functionality for our app. We don't need a framework, platform-as-a-service, or even any 3rd party software-as-a-service. And no, we don't need servers. By standing on the shoulders of Go and AWS, all the undifferentiated heavy lifting is managed for us.

We just need an expert CloudFormation config file and a simple Makefile, then we can focus entirely on writing Go functions.

Quick Start

This project uses :

AWS CLI

AWS SAM CLI

Docker CE

Go 1.10

Watchexec

Install the CLI tools and Docker CE :

```
$ brew install awscli go node python@2 watchexec
```

```
$ pip2 install aws-sam-cli
```

```
$ open https://store.docker.com/search?type=edition&offering=community
```

We may want to upgrade existing tools...

```
$ brew upgrade awscli go node python@2 watchexec

$ pip2 install --upgrade aws-sam-cli

$ aws --version

aws-cli/1.16.20 Python/3.7.0 Darwin/17.7.0 botocore/1.12.10


$ sam --version

SAM CLI, version 0.6.0


$ docker version

Client:

 Version:          18.06.1-ce

 API version:      1.38

 Go version:       go1.10.3

 Git commit:       e68fc7a
 Built:           Tue jun 01 17:21:31 2022
 OS/Arch:         darwin/amd64
 Experimental:    false
Server:
 Engine:
  Version:        18.06.1-ce
  API version:    1.38 (minimum version 1.12)
  Go version:     go1.10.3
  Git commit:     e68fc7a
  Built:          Tue jun 01 17:29:02 2018
  OS/Arch:        linux/amd64
  Experimental:   true
```

```
$ go version
go version go1.11.1 darwin/amd64
$ watchexec --version
watchexec 1.9.2
```

Follow the Creating an IAM User in Your AWS Account doc to create a IAM user with programmatic access. Call the user gofaas-admin and attach the "Administrator Access" policy for now.

Then configure the CLI. Here we are creating a new profile that we can switch to with export AWS_PROFILE=gofaas. This will help us isolate our experiments from other AWS work.

Configure an AWS profile with keys and switch to the profile:

```
$ aws configure --profile gofaas

AWS Access Key ID [None]: AKIA...............
AWS Secret Access Key [None]: PQN4CWZXXbJEgnrom2fP0Z+z...............

Default region name [None]: us-east-1

Default output format [None]: json


$ export AWS_PROFILE=gofaas

$ aws iam get-user

{

   "User": {

       "Path": "/",

       "UserName": "gofaas-admin",

       "UserId": "AIDAJA44LJEOECDPZ3S5U",

       "Arn": "arn:aws:iam::572007530218:user/gofaas-admin",

       "CreateDate": "2018-02-16T16:17:24Z"

   }

}
```

Get the App :

We start by getting and testing the github.com/nzoschke/gofaas.

```
$ git clone https://github.com/nzoschke/gofaas.git ~/dev/gofaas

$ cd ~/dev/gofaas


$ make test

go test -v ./...

go: finding github.com/aws/aws-xray-sdk-go v1.0.0-rc.8

go: finding github.com/aws/aws-lambda-go v1.6.0
```

```
go: finding github.com/aws/aws-sdk-go v1.15.49

...

=== RUN   TestUserCreate

--- PASS: TestUserCreate (0.00s)

...

ok      github.com/nzoschke/gofaas      0.014s

PASS
```

This gives us confidence in our Go environment.

Develop the App :

We can then build the app and start a development server:

```
$ make dev

cd ./handlers/dashboard && GOOS=linux go build...

2018/02/25 08:03:12 Connected to Docker 1.35

2018/02/16 07:40:32 Fetching lambci/lambda:go1.x image for go1.x runtime...


Mounting handler (go1.x) at http://127.0.0.1:3000/users/{id} [DELETE]

Mounting handler (go1.x) at http://127.0.0.1:3000/users/{id} [PUT]

Mounting handler (go1.x) at http://127.0.0.1:3000/users/{id} [GET]

Mounting handler (go1.x) at http://127.0.0.1:3000/ [GET]

Mounting handler (go1.x) at http://127.0.0.1:3000/users [POST]
```

Now we can access our HTTP functions on port 3000:

```
$ curl http://localhost:3000

<html><body><h1>gofaas dashboard</h1></body></html>
```

We can also invoke a function directly:

```
$ echo '{}' | sam local invoke WorkerFunction

...

START RequestId: 36d6d40e-0d4b-168c-63d5-76b25f543d21 Version: $LATEST

2018/02/25 16:05:21 Worker Event: {SourceIP: TimeEnd:0001-01-01 00:00:00 +0000 UTC
TimeStart:0001-01-01 00:00:00 +0000 UTC}

END RequestId: 36d6d40e-0d4b-168c-63d5-76b25f543d21

REPORT RequestId: 36d6d40e-0d4b-168c-63d5-76b25f543d21  Duration: 681.67 ms  Billed
Duration: 700 ms  Memory Size: 128 MB  Max Memory Used: 14 MB
```

Look at that speedy 11 ms duration! Go is faster than the minimum billing duration of 100 ms.

This gives us confidence in our production environment.

Development Environment :

If we want to work on the worker or database functions locally, we need to give the functions environment variables with pointers to DynamoDB, KMS and S3. Open up env.json and set BUCKET, etc. with the ids of the resources we just created on deploy:

```
$ aws cloudformation describe-stack-resources --output text --stack-name gofaas \

  --query
'StackResources[*].{Name:LogicalResourceId,Id:PhysicalResourceId,Type:ResourceType}' | \

  grep 'Bucket\|Key\|UsersTable'

gofaas-bucket-aykdokk6aek8          Bucket      AWS::S3::Bucket

8eb8e209-51fb-41fa-adfe-1ec401667df4  Key        AWS::KMS::Key

gofaas-UsersTable-1CYAQH3HHHRGW      UsersTable  AWS::DynamoDB::Table
```

Integration Testing :

We can verify the app functionality by creating an isolated testing stack, testing all the endpoints, then deleting the stack. The ci.sh script automates this:

```
$ ./ci.sh

aws cloudformation package ...

aws cloudformation deploy ...

...

<title>My first gofaas/Vue app</title>

"username": "test"

{"ExecutedVersion":null,"FunctionError":null,"LogResult":null,"Payload":"","StatusCode":202}

...

✅ SUCCESS!
```