How to Migrate a Go API to Serverless (in Under 10 Mins)

17 April, 2018

I'm going to step you through the process converting an existing Go API to serverless and deploying it to to AWS Lambda & API Gateway with AWS Severless Application Model (SAM). The whole process should take under 10 minutes. Let's get started!

1. Setup

Our example API uses the `HttpRouter` package so let's install that first.

```
$ go get github.com/julienschmidt/httprouter
$ tree
.
├── main.go
└── handlers.go
```

We have a single HTTP handler defined that will return a 200 HTTP response with the body `ok`.

```
# handlers.go
package main

import "net/http"

func HealthHandler(w http.ResponseWriter, r *http.Request)
{
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("ok"))
}
```

Our entrypoint to the application, the `main` function, attaches the `HealthHandler` to the `/healthz` route and listens for HTTP requests on port `8080`.

```
# main.go
package main
```

```
import (
    "fmt"
    "log"
    "net/http"

    "github.com/julienschmidt/httprouter"
)

const (
    serverPort = 8000
)

func main() {
    router := httprouter.New()
    router.Handler("GET", "/healthz",
http.HandlerFunc(goserverlessapi.HealthHandler))

    fmt.Printf("Server listening on port: %d\n",
serverPort)
    log.Fatal(http.ListenAndServe(fmt.Sprintf(":%d",
serverPort), router), nil)
}
```

Let's build and run this locally to check everything's working okay.

```
$ go build -o go-serverless-api && ./go-serverless-api
Server listening on port: 8080
```

2. Convert Application Code to Serverless

In order to deploy to a serverless backend we need to be able to handle requests from AWS Lambda. Lambda functions have a different signature to regular HTTP handlers. Imagine if we had not one but hundreds in our application. We would have to manually update all the functions and re-write tests and in doing so we would forfeit our ability to deploy to non-serverless backends.

There is a solution which avoids all the problems just listed. We will create a modified entrypoint just for AWS Lambda. Using the `gateway` package we will swap out `net/http`'s `ListenAndServe` for `gateway.ListenAndServe` which will convert the payload that AWS Lambda provides into the `*http.Request` type that HTTP handlers accept.

In order to implement this second entrypoint we need to reorganise our project. We will create a directory for the AWS Lambda one and move our original entrypoint to a new folder too.

```
# original entrypoint moved to new location
```

```
$ mkdir -p cmd/go-serverless-api

# new entrypoint for lambda
$ mkdir -p cmd/go-serverless-api-lambda
```

We will copy the `main.go` file into each of these new directories and then remove it from the root of our project.

```
$ cp main.go cmd/go-serverless-api
$ cp main.go cmd/go-serverless-api-lambda
$ rm main.go
$ tree
.
├── cmd
|   ├── go-serverless-api
|   |   └── main.go
|   └── go-serverless-api-lambda
|       └── main.go
└── handlers.go
```

The package in the root of our project is no longer going to be the `main` package (the one Go uses to run your application). We will rename it to `goserverlessapi` so that we can import it as a library into our new entrypoints which will both become `main` packages.

```
$ grep -l 'package main' *.go | xargs sed -i 's/package main/package goserverlessapi/g'
```

After a simple find and replace on the Go files in the root of your project your `handlers.go` should look like this.

```
# handlers.go
package goserverlessapi

import "net/http"

func HealthHandler(w http.ResponseWriter, r *http.Request)
{
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("ok"))
}
```

Now we need to update our original entrypoint to import the `goserverlessapi` package and access the `HealthHandler` function as an export from that. Note that you will need to modify the import path of the `goserverlessapi` package to match that of your project root's location in your `$GOPATH`. The correct path for the example application on github used in this tutorial is `github.com/techjacker/go-serverless-api`.

```
# cmd/go-serverless-api/main.go
package main

import (
    "fmt"
    "log"
    "net/http"

    "github.com/julienschmidt/httprouter"
    "github.com/techjacker/go-serverless-api"
)

const (
    serverPort = 8080
)

func main() {
    router := httprouter.New()
    router.Handler("GET", "/healthz",
http.HandlerFunc(goserverlessapi.HealthHandler))

    fmt.Printf("Server listening on port: %d\n",
serverPort)
    log.Fatal(http.ListenAndServe(fmt.Sprintf(":%d",
serverPort), router), nil)
}
```

Next update the AWS Lambda entrypoint. We are going to use the `gateway` package which is a drop-in replacement for Go net/http when running on AWS Lambda & API Gateway.

```
$ go get github.com/apex/gateway
```

```
# cmd/go-serverless-api-lambda/main.go
package main

import (
    "log"
    "net/http"

    "github.com/apex/gateway"
    "github.com/julienschmidt/httprouter"
    "github.com/techjacker/go-serverless-api"
)

func main() {
    router := httprouter.New()
```

```
    router.Handler("GET", "/healthz",
http.HandlerFunc(goserverlessapi.HealthHandler))
    log.Fatal(gateway.ListenAndServe("", router), nil)
}
```

We have added `gateway` to our imports and swapped it out for `http.ListenAndServe`. The port value is redundant in the Lambda context and the `gateway` package discards it so we can safely remove the port constant and replace it with an empty string. In addition we have included our `HealthHandler` from the `go-serverless-api` package (the package in the root our of project which used to be our main package) as our handler for the `/healthz` path.

3. Build and Run Locally

Let's build and run our original HTTP API again.

```
$ go build \
    -o go-serverless-api \
    ./cmd/go-serverless-api

$ ./go-serverless-api
Server listening on port: 8080
```

Open another terminal window and query it.

```
$ curl -s http://localhost:8080/healthz
ok
```

Everything still works!

Let's do the same with the AWS Lambda version.

```
$ go build \
    -o go-serverless-api-lambda \
    ./cmd/go-serverless-api-lambda

$ ./go-serverless-api-lambda
```

Again, open a new terminal and query it.

```
$ curl -s http://localhost:8080/healthz
http: error: ConnectionError:
HTTPConnectionPool(host='localhost', port=8080): Max
retries exceeded with url: /healthz (Caused by
NewConnectionError(': Failed to establish a new connection:
[Errno 111] Connection refused',)) while doing GET request
to URL: http://localhost:8080/healthz
```

Don't worry this error is expected as the AWS Lambda version is not listening for HTTP connections but instead expects to be fed an `APIGatewayProxyRequest` type.

4. Package Application For AWS Lambda

Build the binary for linux, the operating system that AWS Lambda uses.

```
$ GOOS=linux go build \
    -o go-serverless-api-lambda \
    ./cmd/go-serverless-api-lambda
```

AWS Lambda requires that function code be bundled into a zip so let's go ahead and compress the binary.

```
$ zip go-serverless-api-lambda.zip go-serverless-api-lambda
```

We will be using the created `go-serverless-api-lambda.zip` in the final step - deployment.

5. Deploy

I've seen some tutorials that use the AWS CLI tools to deploy to AWS Lambda using ad hoc commands. This is absolutely the wrong approach to take! You should be automating your infrastructure just like every other aspect of your application. The industry standard for deployment is either Terraform or AWS Cloudformation. Both give you a declarative way to build your infrastructure. You save this configuration in YAML/JSON (Cloudformation) or HCL (Terraform) files which you commit to your repository. The problem with doing things this way is that you have to deal with all of the low level details of the stack. It would be nice if we had a way of describing our infrastructure at a high level in under 20 lines of code instead of hundreds. Enter AWS Severless Application Model (SAM).

AWS Severless Application Model (SAM)

SAM is a new standard spearheaded by Amazon aimed at making deploying serverless infrastructure simpler and more concise. SAM is an open source specification - see the full reference guide. Hopefully other cloud vendors will adopt this in the future and it will become possible to deploy seamlessly to multiple clouds with a single configuration.

Deploy with AWS SAM

Add the following YAML file to the root of your project. This is a SAM template that configures an AWS Lambda Function that runs your Go app and deploys it behind an HTTP interface provided by AWS API Gateway.

```
# template.yaml
```

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: 'Boilerplate Go API.'
Resources:
  GoAPI:
    Type: AWS::Serverless::Function
    Properties:
      Handler: go-serverless-api-lambda
      Runtime: go1.x
      CodeUri: ./go-serverless-api-lambda.zip
      Events:
        Request:
          Type: Api
          Properties:
            Method: ANY
            Path: /{proxy+}
```

The line `Type: AWS::Serverless::Function` creates a Lambda function that is handled (`Handler: go-serverless-api-lambda`) by the binary file we built earlier. This Lambda function can respond to any number of events triggered by other AWS services such as those triggered by AWS S3 and Kinesis. The documentation contains the full list of event of sources. In our case we want it to respond to HTTP requests via API Gateway therefore we set the event to `Type: Api`. SAM implicitly creates an API Gateway for us as part of this which we then configure to respond to any type of HTTP request by setting the method to `ANY`). We tell our API to handle all paths below and including the root by adding `Path: /{proxy+}`.

We still need to upload the zip containing our Go binary to AWS S3. Ensure you have an S3 bucket created ready to receive our zip. This is a one-time operation you'll want to do manually.

```
$ aws s3 mb s3://my-bucket
```

The following command will upload the zip and create a packaged SAM template.

```
$ aws cloudformation package \
    --template-file template.yaml \
    --s3-bucket my-bucket \
    --output-template-file packaged-template.yaml
```

You should now have a `packaged-template.yaml` file pointing to the uploaded zip.

```
# packaged-template.yaml
AWSTemplateFormatVersion: '2010-09-09'
```

```
Transform: 'AWS::Serverless-2016-10-31'
Description: 'Boilerplate Go API.'
Resources:
  GoAPI:
    Type: AWS::Serverless::Function
    Properties:
      Handler: go-serverless-api-lambda
      Runtime: go1.x
      CodeUri: s3://my-bucket/
8982639e71e0d433cd99f9fa4207ecbe
      Events:
        Request:
          Type: Api
          Properties:
            Method: ANY
            Path: /{proxy+}
```

Now let's deploy using this new packaged template.

```
$ aws cloudformation deploy \
    --template-file packaged-template.yaml \
    --stack-name go-serverless-api-stack \
    --capabilities CAPABILITY_IAM
```

The `--capabilities CAPABILITY_IAM` flag is required for cloudformation to create the stack for you as it will involve modifying IAM permissions - AWS mandate you set this explicitly as a safety measure. Under the hood the SAM template is compiled into a regular cloudformation template which is hundreds of lines longer. All of is completely hidden from the user (although you are free to inspect the compiled cloudformation template if you wish).

Setup 6. Test Deployed Serverless API

In order to discover the endpoint of our deployed API we need find out the API Gateway REST `id`.

```
$ aws apigateway get-rest-apis
{
    "items": [
        {
            "id": "0qu18x8pyd",
            "name": "go-serverless-api-stack",
            "createdDate": 1523987269,
            "version": "1.0",
            "apiKeySource": "HEADER",
            "endpointConfiguration": {
                "types": [
                    "EDGE"
```

```
                ]
            }
        }
    ]
}
```

AWS API Gateway addresses take the following format.

`https://<api-rest-id>.execute-api.<your-aws-region>.amazonaws.com/<api-stage>`

A stage is Amazon's term for a deployment. SAM creates two different stages for you: `Stage` and `Prod`. Note the title case which is also used in the URLs! I think AWS forgot that everyone calls their test environment `Staging` not `Stage` but nevermind!

So SAM has set up endpoints for us at the following locations.

`https://0qu18x8pyd.execute-api.eu-west-1.amazonaws.com/Stage`

`https://0qu18x8pyd.execute-api.eu-west-1.amazonaws.com/Prod`

Let's invoke our API.

```
$ curl -s https://0qu18x8pyd.execute-api.eu-west-1.amazonaws.com/Prod
{"message":"Missing Authentication Token"}
```

No need to panic! This is the standard API Gateway error when you make a request to the root resource without defining a handler for it. The only handler we have defined is `/healthz`, so let's try that.

```
$ curl -s https://0qu18x8pyd.execute-api.eu-west-1.amazonaws.com/Prod/healthz
ok
```

Voila! Our API is now being powered by a serverless backend.

Full code for this tutorial available on github.

I'll be posting more articles on Go and Serverless soon. Follow me on twitter or medium to get notified when I do.

BUILDING GO PROJECTS WITH DOCKER ON GITLAB CI

Apr 12, 2018

3 minutes read

Intro

This post is a summary of my research on building Go projects in a Docker container on CI (Gitlab, specifically). I found solving private dependencies quite hard (coming from a Node/.NET background) so that is the main reason I wrote this up. Please feel free to reach out if there are any issues or a submit pull request on the Docker image.

Dep

As dep is the best option for managing Go dependencies right now, the build will need to run `dep ensure` before building.
Note: I personally do not commit my `vendor/` folder into source control, if you do, I'm not sure if this step can be skipped or not. The best way to do this with Docker builds is to use `dep ensure -vendor-only`. See here.

Docker Build Image

I first tried to use `golang:1.10` but this image doesn't have:

- curl
- git
- make
- dep
- golint

I have created my own Docker image for builds (github / dockerhub) which I will keep up to date - but I offer no guarantees so you should probably create and manage your own.

Internal Dependencies

We're quite capable of building any project that has publicly accessible dependencies so far. But what about if your project depends on another private gitlab repository?
Running `dep ensure` locally should work with your git setup, but once on CI this doesn't apply and builds will fail.

Gitlab Permissions Model

This was added in Gitlab 8.12 and the most useful feature we care about is the `CI_JOB_TOKEN` environment variable made available during builds.
This basically means we can clone dependent repositories like so

```
git clone https://gitlab-ci-token:${CI_JOB_TOKEN}
@gitlab.com/myuser/mydependentrepo
```

However we do want to make this a bit more user friendly as dep will not magically add credentials when trying to pull code.
We will add this line to the `before_script` section of the `.gitlab-ci.yml`.

```
before_script:
  - echo -e "machine gitlab.com\nlogin gitlab-ci-
token\npassword ${CI_JOB_TOKEN}" > ~/.netrc
```

Using the `.netrc` file allows you to specify which credentials to use for which server. This method allows you to avoid entering a username and password every time you pull (or push) from Git. The password is stored in plaintext so you shouldn't do this on your own machine. This is actually for `cURL` which Git uses behind the scenes. Read more here.

Project Files

Makefile

While this is optional, I have found it makes things easier. Configuring these steps below means in the CI script (and locally) we can run `make lint`, `make build` etc without repeating steps each time.

```
GOFILES = $(shell find . -name '*.go' -not -path './vendor/
*')
GOPACKAGES = $(shell go list ./...  | grep -v /vendor/)

default: build

workdir:
    mkdir -p workdir

build: workdir/scraper

workdir/scraper: $(GOFILES)
    GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -o
workdir/scraper .
```

```
test: test-all

test-all:
	@go test -v $(GOPACKAGES)

lint: lint-all

lint-all:
	@golint -set_exit_status $(GOPACKAGES)
```

.gitlab-ci.yml

This is where the Gitlab CI magic happens. You may want to swap out the image for your own.

```
image: sjdweb/go-docker-build:1.10

stages:
  - test
  - build

before_script:
  - cd $GOPATH/src
  - mkdir -p gitlab.com/$CI_PROJECT_NAMESPACE
  - cd gitlab.com/$CI_PROJECT_NAMESPACE
  - ln -s $CI_PROJECT_DIR
  - cd $CI_PROJECT_NAME
  - echo -e "machine gitlab.com\nlogin gitlab-ci-
token\npassword ${CI_JOB_TOKEN}" > ~/.netrc
  - dep ensure -vendor-only

lint_code:
  stage: test
  script:
    - make lint

unit_tests:
  stage: test
  script:
    - make test

build:
  stage: build
  script:
    - make
```

What This Is Missing

I would usually be building a Docker image with my binary and pushing that to the Gitlab Container Registry.
You can see I’m building the binary and exiting, you would at least want to store that binary somewhere (such as a build artifact).