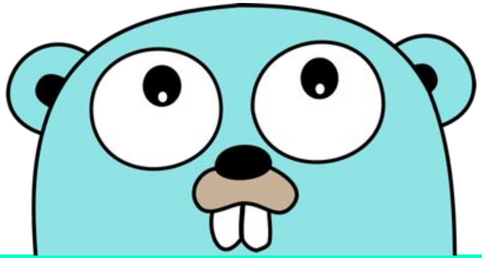
Serverless Go



how to use Go to build serverless services

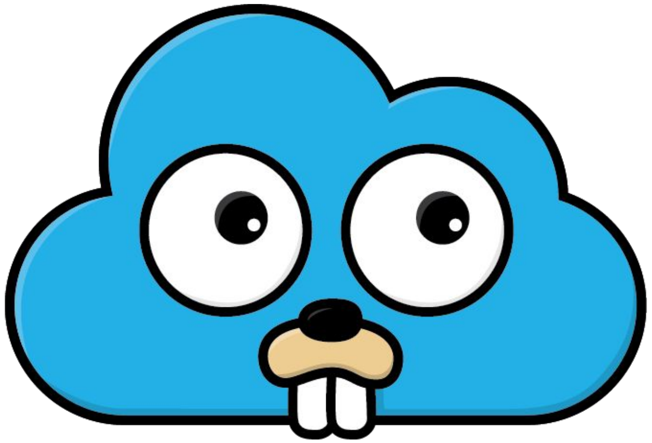
About me

- Software Developer @Easy Network
  - Python & Go
  - Working with AWS Technologies
- Working with Go since 2018
  - AWS Lambda
- Contacts:
  - github.com/ferruvich
  - danieleferru@easy-network.it

Serverless & Function as a Service

Serverless?

- Cloud Computing model
- Abstracts server management
- No infrastructure management
  - Managed Cloud Computing Service
  - External services and API

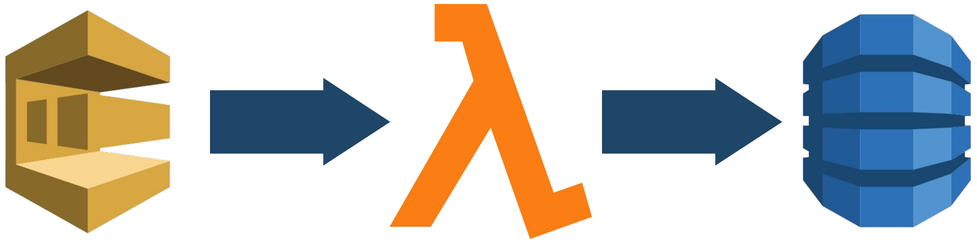


Serverless?

- Developer have only to think about
  - Writing code
  - Solving problems
  - Choosing the correct services
  - Putting them together

FaaS - Function as a Service

- Deploy and invoke **Functions**
- Triggered by **Events**
- That uses **Resources**

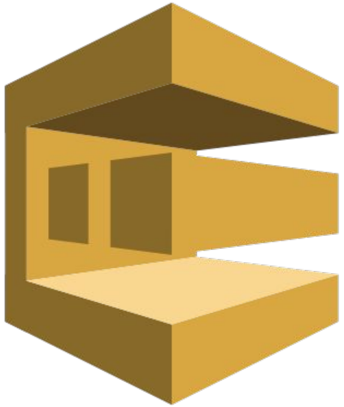


FaaS - Function

- Simple piece of code deployed in the cloud
  - Go binary
- Stateless
  - No state on Function’s server
- Designed to do one thing, and do it well

FaaS - Events

- Provided by the FaaS provider
- They can be of many types:
  - HTTP Requests (*API Gateway*)
  - Queue-Based (*SQS*)
  - Pub-Sub (*SNS*)
  - File upload (*S3 file upload*)
- Event emit => New Function execution process



FaaS - Resources

- Provided by the FaaS provider
- DataBase (DynamoDB, Aurora)
- File Storage (S3)
- Operation on resources can trigger events!

FaaS - Advantages

- Fewer developer logistics
- More time focused on writing code
- Never pay for idle resources
- Inherently scalable

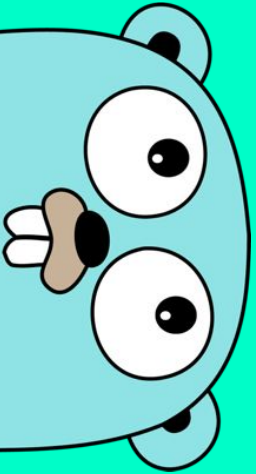
FaaS - Disadvantages

- Decreased transparency
- Potential tough to debug
- Auto-scaling of function calls = auto-scaling of price
- Hard to keep track of so much functions

FaaS - Providers

- Early 2014: **hook.io**
- Nov. 2014: **AWS Lambda**
- 2016: **Azure Functions**
  - March 2016: Preview
  - Fully released on Nov. 2016
- 2017: **Google Cloud Functions**
  - March 2017: Beta
  - Fully released on Oct. 2017

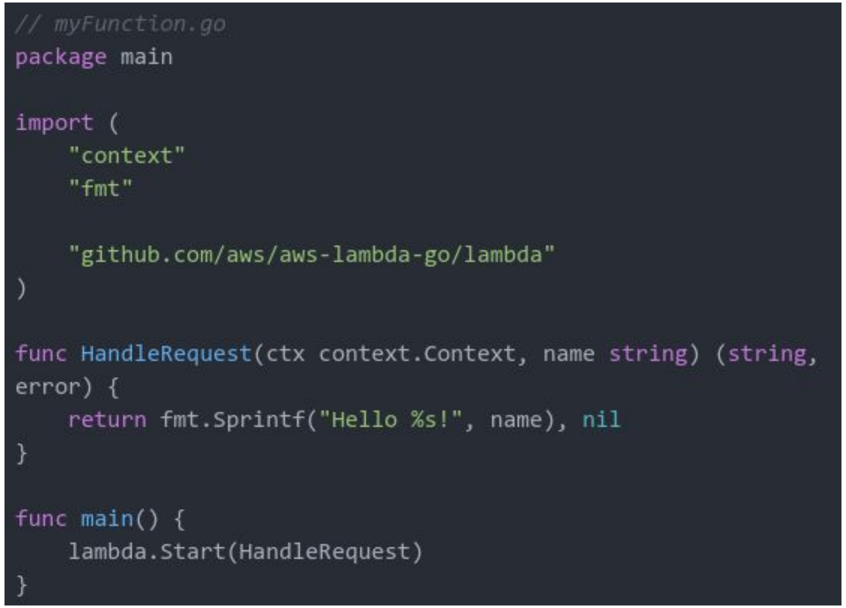
hook.io



Go + Serverless

Golang Functions

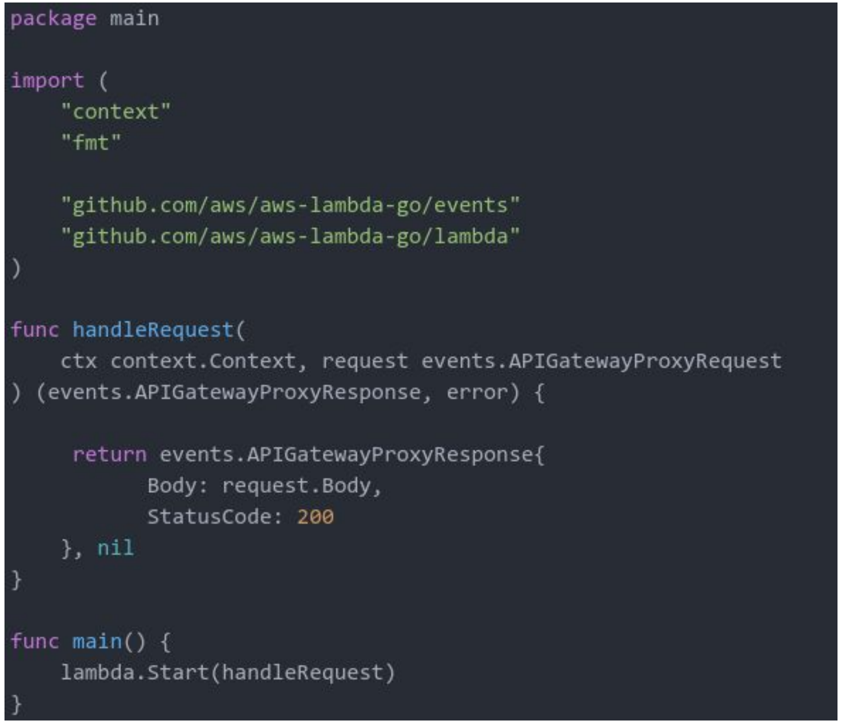
- Do one thing and do it well
- Event as input
- Response or error as output
- Error response handled directly by provider
- Panics handled too



```go
// myFunction.go
package main

import (
    "context"
    "fmt"

    "github.com/aws/aws-lambda-go/lambda"
)

func HandleRequest(ctx context.Context, name string) (string,
error) {
    return fmt.Sprintf("Hello %s!", name), nil
}

func main() {
    lambda.Start(HandleRequest)
}
```

Golang Libraries

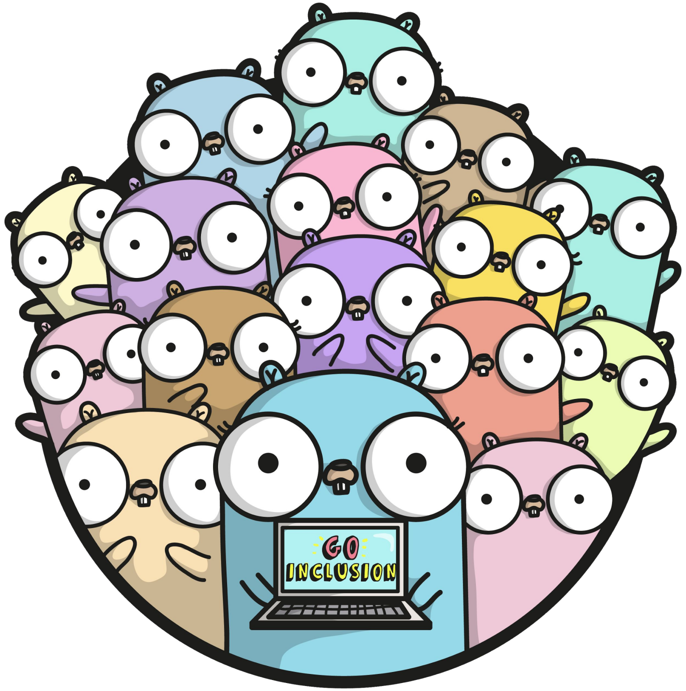
- Ease go Functions development
- Provided by FaaS providers
- Examples
  - aws/aws-lambda-go
  - aws/aws-sdk-go
  - Azure/azure-sdk-for-go
  - googleapis/google-cloud-go



```go
package main

import (
    "context"
    "fmt"

    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
)

func handleRequest(
    ctx context.Context, request events.APIGatewayProxyRequest
) (events.APIGatewayProxyResponse, error) {

     return events.APIGatewayProxyResponse{
           Body: request.Body,
           StatusCode: 200
    }, nil
}

func main() {
    lambda.Start(handleRequest)
}
```

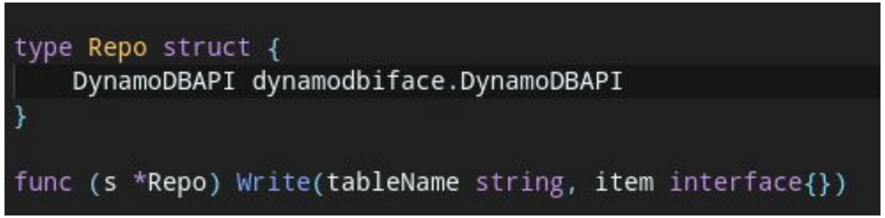
Golang Libraries - Interfaces

- Service API interfaces
- They contain all API definition for a service
- Use only interface dependencies
- They can be used to write mocks for unit testing

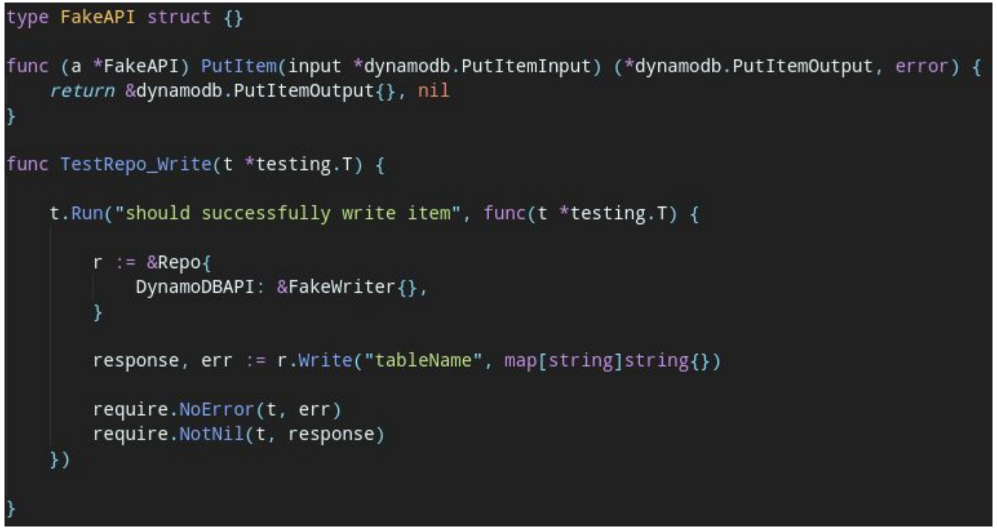


Testing - Interface

- Guarantees to test your code in isolation
- You can mock any wanted behaviour from API



```go
type Repo struct {
    DynamoDBAPI dynamodbiface.DynamoDBAPI
}

func (s *Repo) Write(tableName string, item interface{})
```



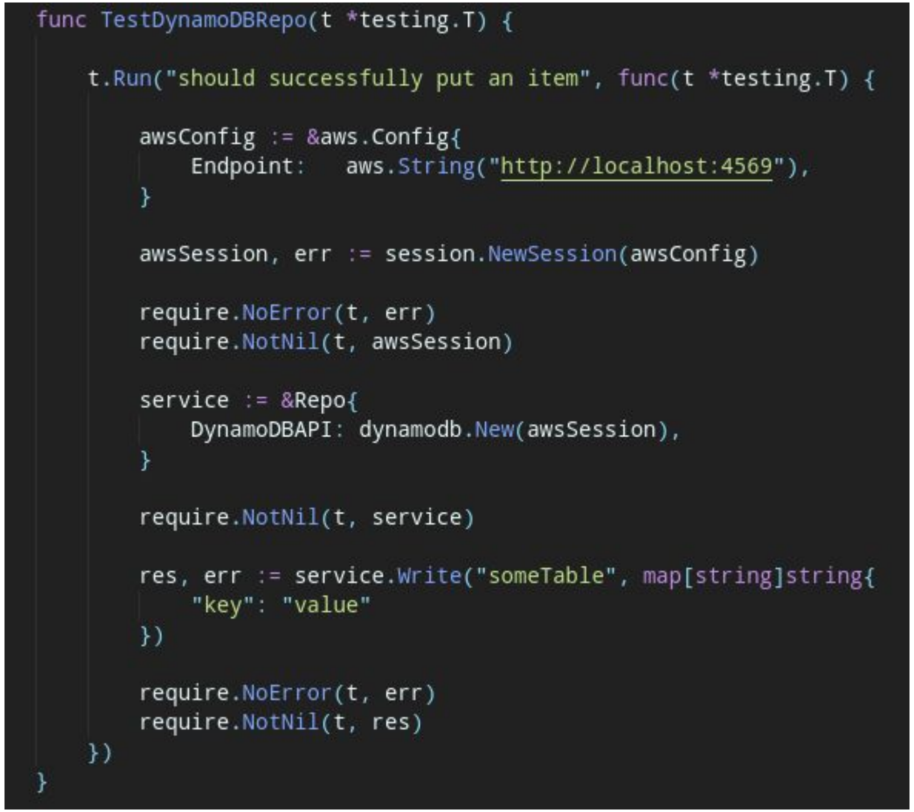
```go
type FakeAPI struct {}

func (a *FakeAPI) PutItem(input *dynamodb.PutItemInput) (*dynamodb.PutItemOutput, error) {
    return &dynamodb.PutItemOutput{}, nil
}

func TestRepo_Write(t *testing.T) {

    t.Run("should successfully write item", func(t *testing.T) {

        r := &Repo{
            DynamoDBAPI: &FakeWriter{},
        }

        response, err := r.Write("tableName", map[string]string{})

        require.NoError(t, err)
        require.NotNil(t, response)
    })

}
```

Testing - Localstack

- Simulates FaaS provider
  - AWS
  - Future development for Azure and GCP
- Serves each mocked service using a specific port
- It can be served using docker easily

Testing - Localstack

- Effectively use aws library functions while testing
- You only need to setup the correct endpoint on the library configuration



```go
func TestDynamoDBRepo(t *testing.T) {

    t.Run("should successfully put an item", func(t *testing.T) {

        awsConfig := &aws.Config{
            Endpoint:   aws.String("http://localhost:4569"),
        }

        awsSession, err := session.NewSession(awsConfig)

        require.NoError(t, err)
        require.NotNil(t, awsSession)

        service := &Repo{
            DynamoDBAPI: dynamodb.New(awsSession),
        }

        require.NotNil(t, service)

        res, err := service.Write("someTable", map[string]string{
            "key": "value"
        })

        require.NoError(t, err)
        require.NotNil(t, res)
    })
}
```

Deploy - The Serverless Framework

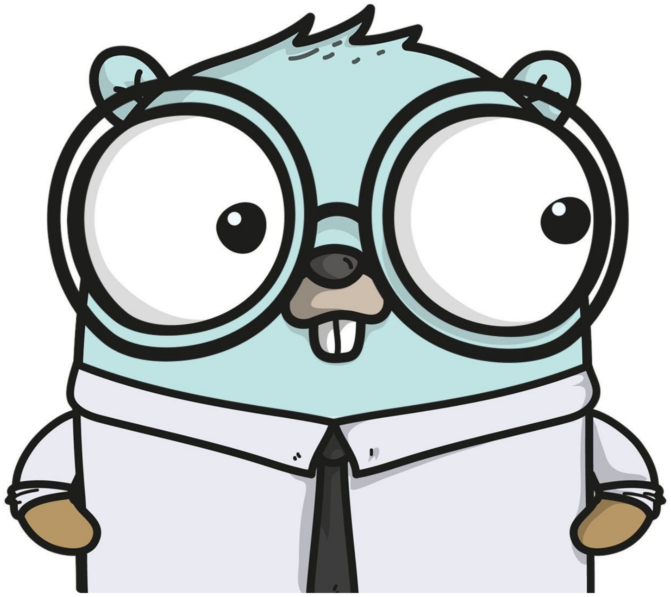
- Deploy Functions and Resources
- Setup Function event triggers
- YAML File
- Command Line Interface (CLI)
- Extendable with plugins

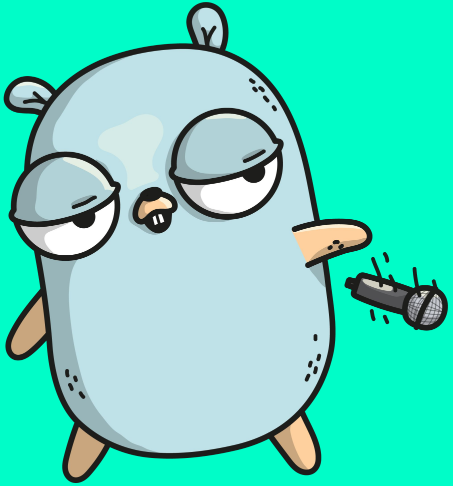
Serverless Framework

- Workflow
  - **create**
    - bootstraps a new project
    - prepared templates
  - *develop*
    - do your stuff
  - **deploy**
    - AWS, GCP, Azure
  - **invoke**
    - locally or remotely
  - **logs**

Today's talk

- Serverless approach and FaaS
- Basic golang tools to handle serverless functions
- Testing tools
  - Library interfaces
  - Localstack
- Deploying tools
  - Serverless





THANK YOU!!