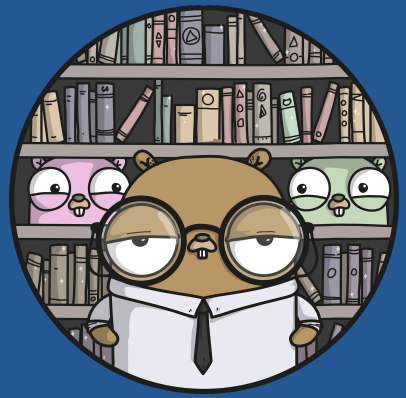
Project Structure & Testing Approach for Maintainable Microservices

Andrea Medda - July 11th 2019

WHOAMI

- Software Engineer @Curve
  - Dealing daily with distributed systems
  - Build & Maintain Go microservices
- Gopher since early 2017
  - Approached Go for fun and University
  - Introduced Go in @Easynetwork
- Open Source contributor
  - github.com/andream16
- andrea.medda@imaginecurve.com

Agenda



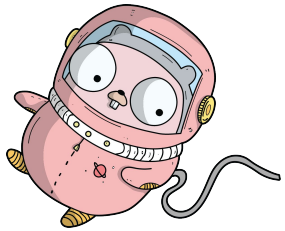
- Challenges of Microservices in Go
  - build, test, monitor & maintenance
- Hexagonal Architecture in Go
  - smallest component of Go
- Testing & Interfaces
  - unit testing, mocking, e2e, integration

What's a microservice?

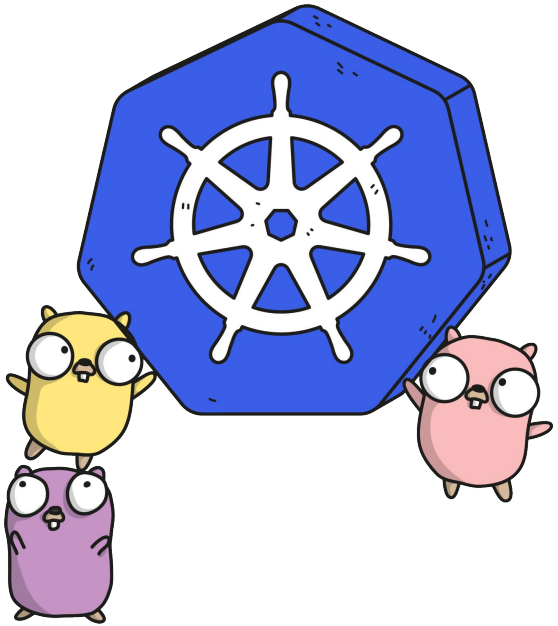
Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities
- Owned by a small team

Definition from microservices.io



Microservices in Go



- Small services
  - small binary
  - easy concurrency
- Easy to deploy
  - Kubernetes, Lambdas, ...
- Easy to monitor
  - CPU, Memory, APM, Status Codes, ...

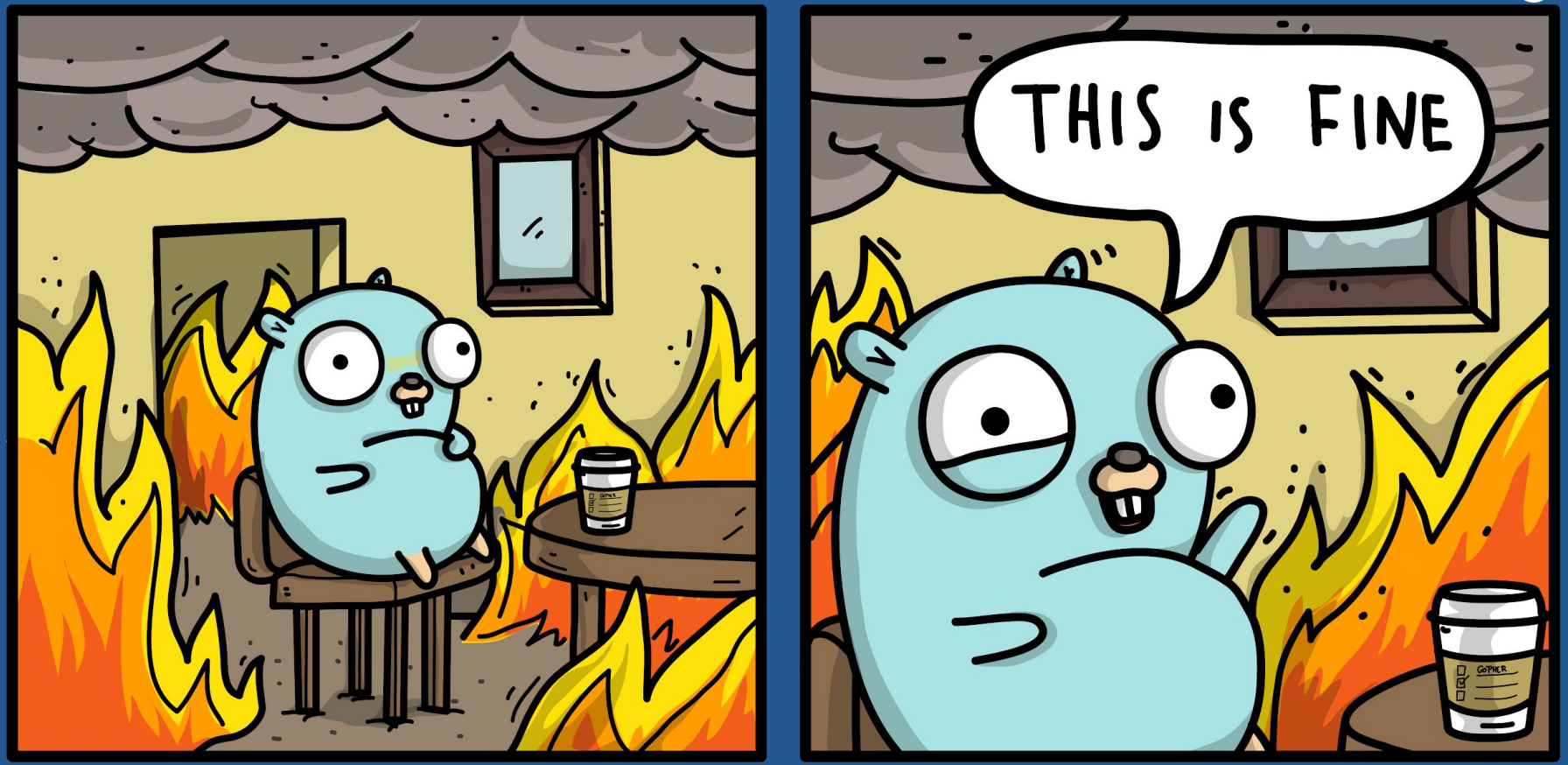
Microservices in Go - Problems

- Multiple teams, multiple projects
  - different project structure
  - different testing approach
  - dependencies between services

- Frameworks
  - big binaries
  - know how to use the FW rather than following good programming practices (although they can be helpful)

Microservices in Go - Even more Problems

- Coupling - not depending on abstractions
  - hellish to maintain
- Shared state code and tests!
  - very difficult to test and maintain
- Multiple stores, transport layers
  - mongo + postgresql ?
  - REST & RabbitMQ ?


THIS IS FINE
GOPHER

How to deal with go microservices

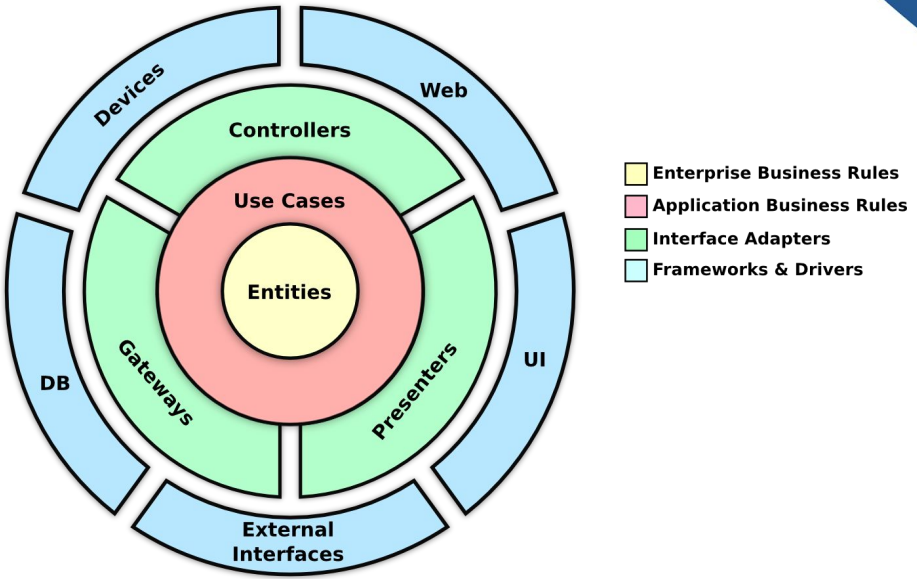
- Adopt clean code practices
  - adopt concept of the go package
  - interfaces are your friend
  - build very small, testable, reusable and optional packages

- Tame shared state
  - make everything explicit
  - consume packages in one place
  - avoid init functions and global variables

- Keep everything unexported until you really need it to be public
  - embrace internal/ and cmd/

Hexagonal Architecture

- DDD & clean code practice
- semantically close concepts live in different and independent “layers”
- layers depend only on abstractions
- maintainable & reusable components

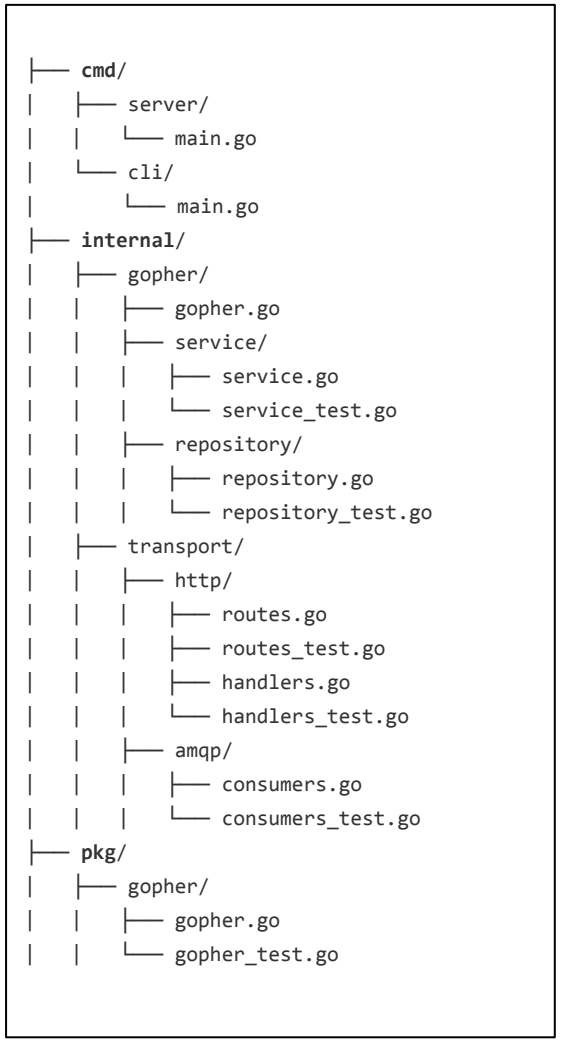


Hex Architecture in a Go microservice

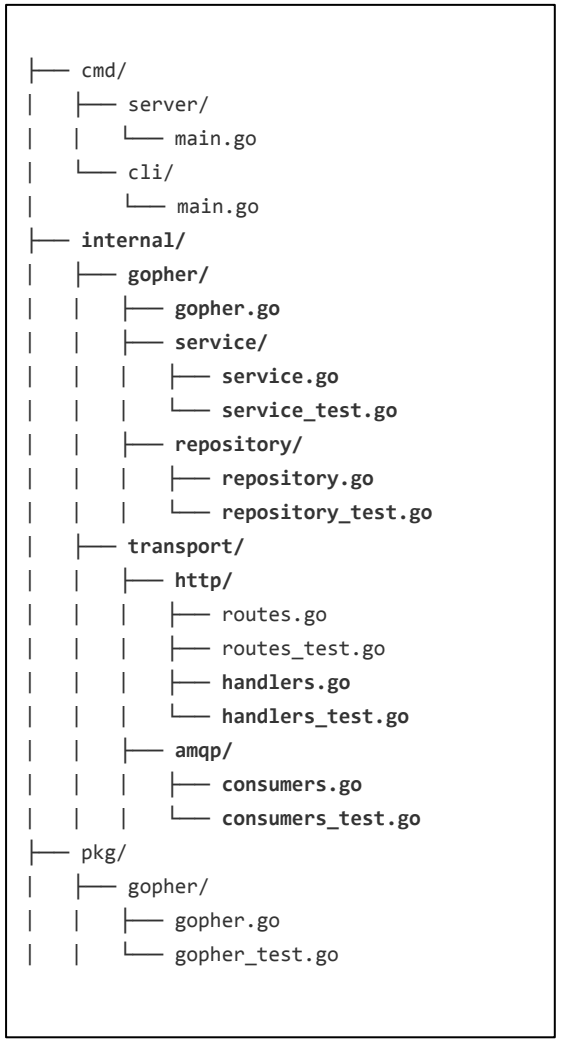
Main components:

- **Model** - entity on which we base the business logic
- **Service** - where the business logic lives, doesn't depend on anything else
- **Handler** - the way of communicating with a user
- **Repository** - database adapter, doesn't depend on anything else

Hex Project Structure - Basic



- cmd/
  - consume internal packages explicitly in one place for each binary
- internal/
  - everything about your service that shouldn't be exported
- pkg/
  - service client, code that is exposed and can be safely imported in other services



Hex Project Structure - Hex

- Model
  - Gopher entity
- Service
  - business logic regarding Gopher
- Repository
  - database wrapper for the domain
- Handlers
  - http & amqp handlers

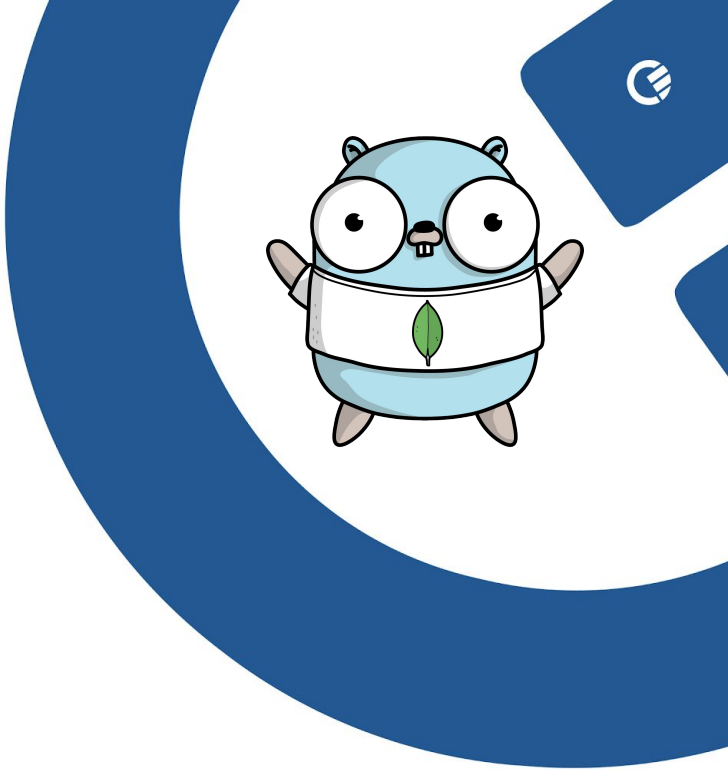
Go-Hex Entity

- An entity should contain only go types.
- Do not add receiver methods or other functions
- Do not add domain constants and variables

```
package gopher

type Gopher struct {
        Name string `json:"gopher"`
        Color string `json:"color"`
}
```

Go-Hex Repository

```
package repository

import (
        “context”

        "go.mongodb.org/mongo-driver/mongo"
        "go.mongodb.org/mongo-driver/mongo/options"
)

type Repositorer interface {
        Create(ctx context.Context, g *gopher.Gopher) error
        Find(ctx context.Context, name string) (*gopher.Gopher, error)
}

type Repository struct {
        client *mongo.Client
}

func (r *Repository) Create(ctx context.Context, g *gopher.Gopher) error { … }

func (r *Repository) Find(ctx context.Context, name string) (*gopher.Gopher, error) { … }

func New(ctx context.Context, opts …*options.Option)  (*Repository, error) { … }
```

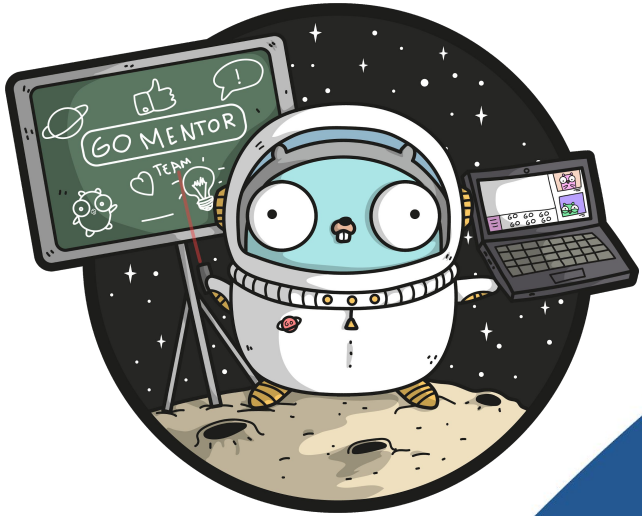


- Wrapper of a database
- Expose an interface and a type that implements such methods

Go-Hex Service

```
package service

import (
        “context”

        “github.com/andream16/gophers/internal/gopher”
        “github.com/andream16/gophers/internal/logger”
        “github.com/andream16/gophers/internal/tracker”
)

type Servicer interface {
        Create(ctx context.Context, g *gopher.Gopher) error
        Find(ctx context.Context, name string) (*gopher.Gopher, error)
}

type Service struct {
        repository Repositorer
        logger logger.Logger,
        tracker tracker.Tracker
}

func (svc *Service) Create(ctx context.Context, g *gopher.Gopher) error { ... }

func (svc *Service) Find(ctx context.Context, name string) (*gopher.Gopher, error) { … }

func New(repository Repositorer, logger logger.Logger, tracker tracker.Tracker)  *Service { … }
```

- Contains business logic
- Only interface dependencies. Concrete implementations are passed directly in the main



Go-Hex Handler

- Only service interface dependency
- GRPC/HTTP
- No need to re-implement service layer

```
package http

import (
        “context”
        “net/http”

        “github.com/andream16/gophers/internal/gopher/service”
)

type Handler struct {
        service gopher.Servicer
}

func (h *Handler) Create(w http.ResponseWriter, r *http.Request) { ... }

func  (h *Handler) Find(w http.ResponseWriter, r *http.Request) { ... }

func NewHandler(svc service.Service) *Handler  { … }
```

Packages and Interfaces

- a package is the smallest go component
- expose small interfaces that describe the functionality
- interfaces allow us to depend on abstractions, to unit test easily and generate mocks
- put all the pieces together in one place; consume all your packages in the main file

Put all the pieces together

```
package main

import (
        ....
)

func main() {
        …
        l, err := logger.New()
        if err != nil {
                log.Fatal(err)
        }
        r, err := repository.New(ctx, nil)
        if err != nil {
                l.Error(err)
        }
        svc, err := service.New(r, l, nil)
        if err != nil {
                l.Error(err)
        }
        h, err := http.NewHandler(r, l, nil)
        if err != nil {
                l.Error(err)
        }
        // start server & attach handler
}
```

- compose your “services” in explicit mains
- avoid initializing components in other packages - make it clear what your program needs to run
- avoid init functions - they are not constructors!
- Keep It Simple Stupid

Test packages

- Test only what can be imported by other packages. Test ALWAYS from POV of a third party user
- Test in isolation. Test only your component. Mock external ones if you need to
- Build your packages in a way that is easy to assert if a certain error occurred - Sentinel Errors

Sentinel and Context'd Errors - Code

```
package service

import (
        …
        “github.com/pkg/errors”
)

var (
        ErrNotFound = errors.New(“gopher_not_found”)
        ErrRepository = errors.New(“repository_err”)
)

func (svc *Service) Find(
        ctx context.Context,
        name string,
) (*Gopher, error) {
        g, err := svc.repository.Find(ctx, name)
        if err != nil {
                if repository.ErrNotFound == errors.Cause(err) {
                        return nil, ErrNotFound
                }
                return nil, errors.Wrap(ErrRepository, err.Error())
        }
        return g, nil
}
```

Sentinel and Context'd Errors - Test

```
package service_test

import (
	...
	“github.com/andream16/gophers/internal/gopher/service”
	“github.com/andream16/gophers/internal/gopher/repository”
)

type mockRepositoryNotFound struct {}
func (m *mockRepositoryNotFound) Find(context.Context, string) (*gopher.Gopher, error) {
	return nil, repository.ErrNotFound
}
type mockRepositoryError struct {}
func (m *mockRepositoryError) Find(context.Context, string) (*gopher.Gopher, error) {
	return nil, errors.New(“someError”)
}

func TestService_Find(t *testing.T) {
	t.Run(“should return Not Found Error because no Gopher was found in the repository”, func(t *testing.T) {
		svc := service.New(
			&mockRepositoryNotFound{},
			nil,
			nil,
		)
		g, err := svc.Find(context.Background(), “puffyGopher”)
		if service.ErrNotFound != errors.Cause(err) {
			t.Fatalf(“expected ErrNotFound, got %s”, err.Error())
		}
		if g != nil {
			t.Fatal(“expected gopher to be nil”)
		}
	})
	...
}
```

Tips on tests

- Don’t share states in tests, make a new t.Run() for each case. Repeating some code is better than sharing.
- Test all mockable or sensed “paths”
- If you need to understand if a component is correctly communicating with an external one (database, third party service) write an integration tests (docker, docker-compose, ...)
- Perfect candidates for integration tests are repository and pkg/client/ layers

What we've talked about

- Build your services in a maintainable manner using a proper project structure
- Avoid global state in all of its shapes
- Test packages, don't test unexported components
- Test in isolation, test the behaviour of your component and not of its dependencies
- Export something only if you really need to

Resources

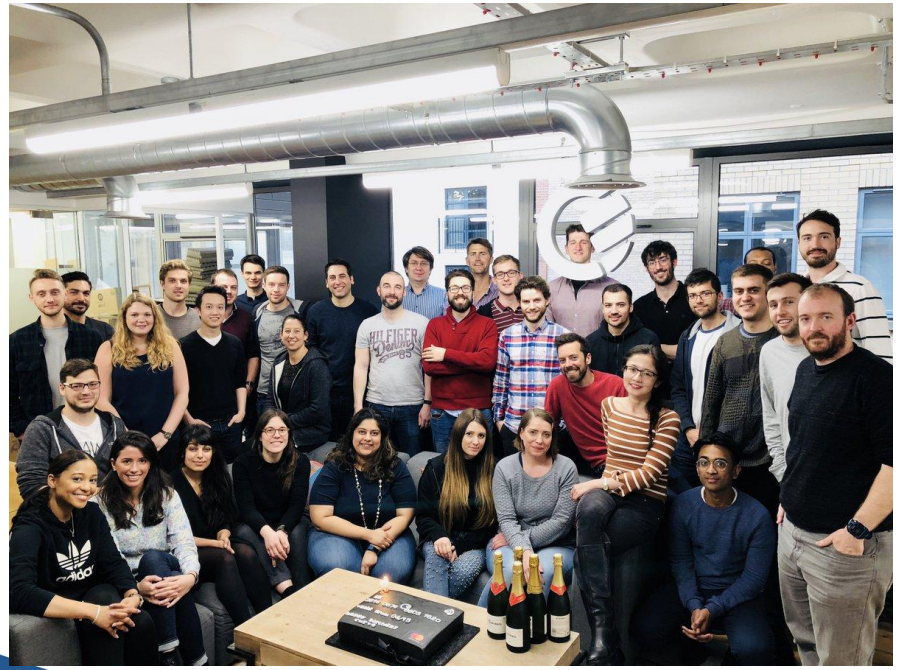
- Gophers stolen from
  github.com/ashleymcnamara/gophers
  @Ashley McNamara

- Hex and maintainable go code sources
  peter.bourgon.org @Peter Bourgon

- Go packages and smallest component of go
  github.com/rakyll @Jaana B. Dogan

We are Hiring!

We're growing very fast, we are ~150 now and we need you!



You'll have the chance to work on something that matters with very interesting tech:

- Go, Kubernetes, Docker, Ambassador, Istio and Prometheus to name a few

You'll also:

- Have an impact on the vibrant Fintech community in London
- Be respected and valued
- Improve quickly

Available on the App Store

GET IT ON Google Play

CURVE

Trustpilot ★★★★★

www.curve.app