Cloud Native Go

Building Scalable, Resilient Microservices for the Cloud in Go

Cloud Native Go

Building Scalable, Resilient Microservices for the Cloud in Go

Agenda

1. Introduction
2. Boring Stuff
3. Live Demos
4. Q & A

Introduction

- Wrote a few books (2 fantasy, 15+ .NET/C#, Go)
- Maker, Tinkerer, Linguist
- Taught people how migrate/build *cloud native* for Pivotal
- Lead Software Engineer for Capital One
- Twitter: **@KevinHoffman**, github **autodidaddict**

*Go to **gopherize.me** for your Gopher Avatar*

Why Go?

- Fast
- Low memory, CPU, and Disk footprint
- Docker images from `SCRATCH`
- Fit more containers per node/VM
- Single, no-dependency binary
- Beautiful, simple language
- Short learning curve

What is Cloud Native?

- API First
- Dependency Management
- Design, Build, Release, Run
- Config, Credentials, and Code
- Logs
- Disposability
- Backing Services
- Environment Parity
- Administrative Processes
- Port Binding
- Stateless Processes
- Concurrency
- Telemetry
- Authentication and Authorization

Microservices Shopping List

- HTTP Server
- Routing
  - URL path variables
  - Query strings
- JSON Encode/Decode
- Middleware (logging, security, etc)

HTTP Server

```
package main

import (
...
)

func main() {
    port := os.Getenv("PORT")
    if len(port) == 0 {
        port = "8080"
    }

    mux := http.NewServeMux()
    mux.HandleFunc("/", hello)

    n := negroni.Classic()
    n.UseHandler(mux)
    hostString := fmt.Sprintf(":%s", port)
    n.Run(hostString)
}

func hello(res http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(res, "Hello from Go!")
}
```

Routing

Routing with Gorilla Mux

```
r := mux.NewRouter()
r.HandleFunc("/products/{key}", ProductHandler)
r.HandleFunc("/articles/{category}/", ArticlesCategoryHandler)
r.HandleFunc("/articles/{category}/{id:[0-9]+}", ArticleHandler)
```

```
s := r.PathPrefix("/zombies").Subrouter()
s.HandleFunc("/", ZombiesHandler)
s.HandleFunc("/sightings", NewSightingHandler).Methods("POST")
s.HandleFunc("/{key}", QueryZombieHandler).Methods("GET")
```

URLs:

- `/zombies`
- `/zombies/sightings` (POST)
- `/zombies/bob` (GET)

Routing

Accessing Route Data

```
vars := mux.Vars(req)
zombieID := vars["zombie-key"]
```

Building URLs

```
r := mux.NewRouter()
r.HandleFunc("/articles/{category}/{id:[0-9]+}", ArticleHandler).
  Name("article")
...
url, err := r.Get("article").URL("category", "technology", "id", "42")
```

Builds - `/articles/technology/42`

JSON Marshaling

Read JSON from Body:

```
payload, _ := ioutil.ReadAll(req.Body)
var newMatchRequest newMatchRequest
err := json.Unmarshal(payload, &newMatchRequest)
```

Write JSON to Response:

```
var mr newMatchResponse
mr.copyMatch(newMatch)
w.Header().Add("Location", "/matches/"+newMatch.ID)
formatter.JSON(w, http.StatusCreated, &mr)
```

Middleware

```
apiRouter := mux.NewRouter()
apiRouter.HandleFunc("/api/post", apiPostHandler(formatter)).Methods("POST")

router.PathPrefix("/api").Handler(negroni.New(
    negroni.HandlerFunc(isAuthorized(formatter)),
    negroni.Wrap(apiRouter),
))
```

```
func isAuthorized(formatter *render.Render) negroni.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request, next http.HandlerFunc) {
        providedKey := r.Header.Get(APIKey)
        if providedKey == "" {
            formatter.JSON(w, http.StatusUnauthorized, struct{ Error string }{"Una
        } else if providedKey != apikey {
            formatter.JSON(w, http.StatusForbidden, struct{ Error string }{"Insuff
        } else {
            next(w, r)
        }
    }
}
```

Shopping List

- ✅ HTTP Server
- ✅ Routing
  - ✅ URL path variables
  - ✅ Query strings
- ✅ JSON Encode/Decode
- ✅ Middleware (logging, security, etc)

We're Doing This All Wrong

Do One Thing Great

A fox knows many things, but a hedgehog one important thing - Archilochus

Hexagonal Architecture

Ports and Adapters

Ports and Adapters in Go

- Build your Core Functionality First
- Decouple HTTP, JSON, Headers, Security from Core Function
- Should be able to *test your core function in isolation*
- Code is *blissfully ignorant* of source and nature of inputs and ultimate destination of replies

Ports & Adapters in a µService Ecosystem

Ports & Adapters in a µService Ecosystem

- API Gateways

Ports & Adapters in a µService Ecosystem

- API Gateways
- Aggregator Services

Ports & Adapters in a µService Ecosystem

- API Gateways
- Aggregator Services
- Message Brokers
  - Topics
  - Pub/Sub

Ports & Adapters in a µService Ecosystem

- API Gateways
- Aggregator Services
- Message Brokers
  - Topics
  - Pub/Sub
- ACL / Translators

Ports & Adapters in a µService Ecosystem

- API Gateways
- Aggregator Services
- Message Brokers
  - Topics
  - Pub/Sub
- ACL / Translators
- Dynamic Service Discovery

Ports & Adapters in a µService Ecosystem

- API Gateways
- Aggregator Services
- Message Brokers
  - Topics
  - Pub/Sub
- ACL / Translators
- Dynamic Service Discovery
- External Configuration

Ports & Adapters in a µService Ecosystem

- API Gateways
- Aggregator Services
- Message Brokers
  - Topics
  - Pub/Sub
- ACL / Translators
- Dynamic Service Discovery
- External Configuration
- *Adapter paths* through system
  - Gateways, Proxies
  - Translators on Data Streams / Queues

Ports & Adapters in a µService Ecosystem

- API Gateways
- Aggregator Services
- Message Brokers
  - Topics
  - Pub/Sub
- ACL / Translators
- Dynamic Service Discovery
- External Configuration
- *Adapter paths* through system
  - Gateways, Proxies
  - Translators on Data Streams / Queues
- Hexagonal arch inside and outside of services

The Shopping List We Ignore

- Security
- Monitoring
- Log Aggregation
- Zero Downtime Deployment
- Build/Deploy Automation
- Tracing
- Metrics
- Audit
- Automatic / Dynamic Horizontal Scaling
- Configuration
- Secrets Management
- Discovery

Starting at the Core

- Define what we're building
- Define why we're building it
- Who are we building it for?
- *Build the core*
- Add onion layers around the core
  - Ports and Adapters
- Remember we're building an ecosystem, not *hello world*

The Go Shopping Sample

Go Shopping Sample - Tech

- `go micro` Framework
  - https://micro.mu/
- Ports and Adapters are *services*
  - Not boilerplate inside service
- Ecosystem-aware from bottom-up
- Dynamic Service Discovery
- gRPC Transport
- RESTful Aggregate API
- ... and much more

Framework vs Library

- Opinionated vs boilerplate
- Spend boilerplate to insulate from opinions
  - Mitigate ceremony with code generation?
- Go-kit - suite of libraries, incur BP cost
- Go-micro - opinionated, less BP
  - Spring Boot - example of opinionated framework

API First

API First is **NOT**:

- RESTful Endpoint First
- Swagger-First
- One API to Rule them All

API First is:

- A strict, *semantically versioned*, interaction specification.
- A team and organization-wide discipline
- Different consumers, channels may need different APIs
  - Ports and Adapters

Defining the Warehouse Service API

- Protocol Buffer IDL
- gRPC Service

```
syntax = "proto3";

package warehouse;

service Warehouse {
    rpc GetWarehouseDetails(DetailsRequest) returns (DetailsResponse);
}

message DetailsRequest {
    string sku = 1;
}

message DetailsResponse {
    WarehouseDetails details = 1;
}

message WarehouseDetails {
    string sku = 1;
    uint32 stock_remaining = 2;
    string manufacturer = 3;
    string model_number = 4;
}
```

Service Implementation

```
package service

import (
    "github.com/autodidaddict/go-shopping/warehouse/proto"
    "golang.org/x/net/context"
)

type warehouseService struct{}

// NewWarehouseService returns an instance of a warehouse handler
func NewWarehouseService() warehouse.WarehouseHandler {
    return &warehouseService{}
}

func (w *warehouseService) GetWarehouseDetails(ctx context.Context,
  request *warehouse.DetailsRequest,
  response *warehouse.DetailsResponse) error {

    response.Manufacturer = "TOSHIBA"
    response.ModelNumber = "T-1000"
    response.SKU = request.SKU
    response.StockRemaining = 35
    return nil
}
```

Let's do some Live Demos

What could possibly go wrong?

Lessons Learned

- Question Everything
- Beware of Protobuf Default Values
- Service-first development
- Code is the *easiest* and *simplest* part of Microservices
  - *NFRs are everything* - logging, metrics, alerts, discovery, health, autoscale, async messaging, etc.
- Decide where you want to be on the opinion vs boilerplate curve
- Go is an *ideal* microservices development language

Next Steps

- Create Services
- Test everything
- Automate everything
- Deploy *all the time*
- Try out `go-micro`
- Try out `go-kit`
- Go Shopping http://github.com/autodidaddict/go-shopping
- These slides https://github.com/autodidaddict/cng-presentation

Q & A