LEAN & MEAN - GO MICROSERVICES WITH DOCKER SWARM AND SPRING CLOUD

ERIK LUPANDER

2017-01-25 I CALLISTAENTERPRISE.SE

CALLISTA
– ENTERPRISE –

What Go?

Go

ON THE AGENDA…

- Background: The footprint problem.
- The Go programming language and developing in Go.
- Go in the context of microservices, Spring Cloud/Netflix OSS and Docker Swarm.
- Demos!

Can Go help us help us reduce the footprint of a microservice?

THE FOOTPRINT PROBLEM

- As Björn just showed us, JVM-based solutions comes with a hefty footprint.
- If you need to run tens or even hundreds of microservice instances, cost is definitely a factor.
- For microservices, some other alternatives are NodeJS, C, C++, Python, Ruby and C#.
  - Very interesting topic, we can return to it over beer tonight…

The Go Language

THE GO LANGUAGE

”Go is an attempt to combine the ease of programming of an interpreted, dynamically typed language, with the efficiency and safety of a statically typed, compiled language.”

Go official FAQ

Go was designed …

THE GO LANGUAGE

”… to eliminate the slowness and clumsiness of software development at Google”

Go official FAQ

WHAT WAS FIXED?

- 50x build time improvement over C++
  - Internal C++ application builds taking 30-75 minutes.
- Language level concurrency
- Better dependency management
- Cross-platform builds
- Readable and maintainable code
  - Even for non superstar developers

THE GO LANGUAGE

- Claims to be
  - efficient, scalable and productive.
- Designed
  - to improve the working environment for its designers and their coworkers.
  - for people who write—and read and debug and maintain—large software systems.
- Is not
  - a research language.

THE GO LANGUAGE

- Go is
  - compiled, statically typed, concurrent, garbage-collected
- Has
  - structs, pointers, interfaces, closures
- But does not have
  - classes, inheritance, operator overloading, pointer arithmetic

What does actual developers think about Go?

”… a disservice to intelligent programmers”

Gary Willoughby - blogger

”… stuck in the 70’s”

Dan Given

”… psuedointellectual arrogance of Rob Pike and everything he stands for”

Keith Wesolowski

But also

”I like a lot of the design decisions they made in the [Go] language. Basically, I like all of them.”

Martin Odersky, creator of Scala

”Go isn’t functional, it’s pragmatical. ”

Frank Mueller, tech blogger

”Go isn’t a very good language in theory, but it’s a great language in practice, and practice is all I care about”

anonymous hackernews poster

Some pros and cons

DEVELOPMENT IN GOLANG - PROS

- Easy to learn, readable, productive and pretty powerful.
- The built-in concurrency is awesome.
- Cross-platform.
- Rich standard APIs and vibrant open source community.
- Quick turnaround and decent IDE support (getting better!)
- Nice bundled tools.
  - Built-in unit testing, profiling, coverage, benchmarking, formatting, code quality…
- Strongly opinionated.
  - Code formatting, compile errors on typical warnings.

DEVELOPING IN GOLANG - SOME CONS

- Missing generics and more powerful built-in collection types.
- Dependency versioning
- Verbose
  - Error checking, no autoboxing of primitive types etc.
- Unit testing and Mocking isn't very intuitive
  - But pretty powerful once one gets the hang of it.

WHO USES GOLANG

- Some well-known software built entirely in golang
  - Docker
  - Kubernetes
  - etcd
- Popularity rankings
  - #13 on Tiobe Index per january 2017, up from #50, largest increase during 2016.

Two code samples

SAMPLE CODE 1 - HELLO WORLD

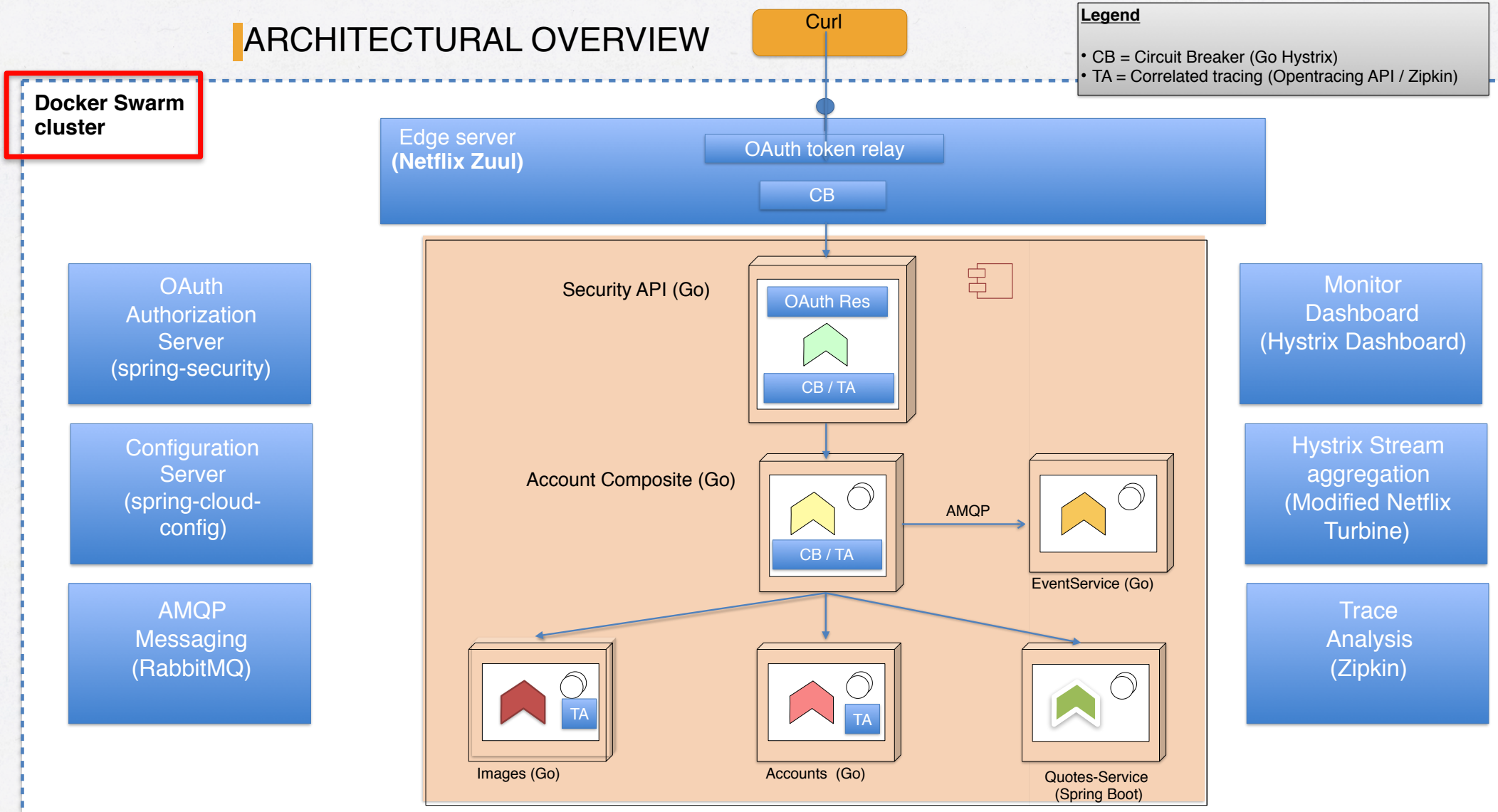
```
package main

import "fmt"

func main() {
        fmt.Println("Hello world!")
}
```

SAMPLE CODE 2 - CONCURRENCY

```
func main() {
        responseChannel := make(chan []byte)

        urls := []string{"http://gp.se", "http://dn.se"}
        for _, url := range urls {
                go doReq(url, responseChannel)    // Make async HTTP call
        }

        for i := 0; i < len(urls); i++ {
                data := <- responseChannel        // Blocks here
                fmt.Println(string(data))
        }
}

func doReq(url string, responseChannel chan []byte) {
        resp, _ := http.Get(url)
        body, _ := ioutil.ReadAll(resp.Body)
        responseChannel <- body                   // Pass result to channel
}
```

Go microservices

ARCHITECTURAL OVERVIEW



WHY GOLANG - RUNTIME CHARACTERISTICS

- Low memory usage
- Overall performance on par with Java (as per Go 1.7)
- Fast startup
- However, Garbage Collector doesn't have ~20 years of maturity and isn't very configurable.

GO MICROSERVICES - STATICALLY LINKED BINARIES

- Statically linked binary produces an executable without external dependencies.
  - No more jar- or dll-hell
  - No requirement on the OS having a JRE or other libraries (except libc)
- Small executable
  - Typically executable size for my microservices is 8-20 mb

DOCKER CONTAINERS & STATICALLY LINKED BINARIES

- In the context of Docker Containers, the statically linked binary allows use of very bare parent images.
- I'm using *iron/base* which is ~6 mb, *alpine* is another popular choice.

```
FROM iron/base

EXPOSE 6868
ADD eventservice-linux-amd64 /
ADD healthcheck-linux-amd64 /

HEALTHCHECK CMD [”./healthcheck-linux-amd64”, ”-port=6868”]

ENTRYPOINT ["./eventservice-linux-amd64", ”-profile=test”]
```

Demo 1
Footprint @ Docker Swarm

GO MICROSERVICE CONSIDERATIONS?

- Microservices doesn’t exist in isolation.
- A pleasant programming language and awesome runtime characteristics isn’t quite enough.
- We need to integrate with various supporting services.

Consider:

MICROSERVICE CONSIDERATIONS

- Centralized configuration
- Service Discovery
- Logging
- Distributed Tracing
- Circuit Breaking
- Load balancing
- Edge
- Monitoring
- Authentication and Authorization

MICROSERVICE CONSIDERATIONS

- On the application level, also consider things like:
  - HTTP / REST / RPC APIs
  - Messaging APIs
  - Persistence APIs
  - Testability
- DevOps

MICROSERVICES - GO VS SPRING BOOT

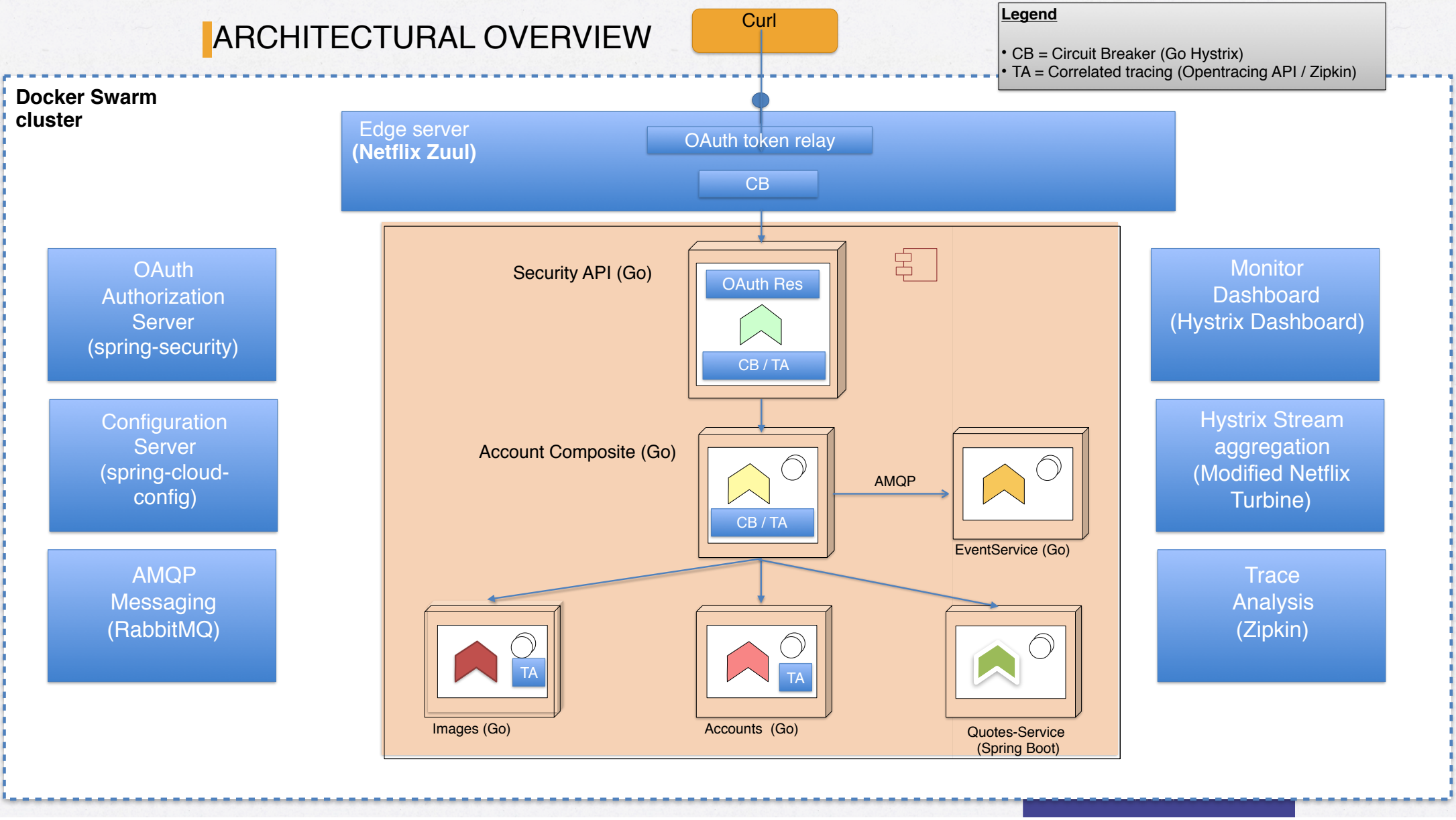
- Spring Cloud / Netflix OSS with Spring Boot microservices provides a very streamlined annotation-driven configuration for integrating with these supporting services.
  - Though the configuration files might be a bit complex.

```
@Autowired
@Named("loadBalancedRestTemplate")
private RestOperations restTemplate;

@HystrixCommand(fallbackMethod = "service1Fallback")
public ResponseEntity<String> callService1(String id) { return callService( url: "http://service-1/r

@HystrixCommand(fallbackMethod = "service2Fallback")
public ResponseEntity<String> callService2(String id) { return callService( url: "http://service-2/r
```

MICROSERVICES - GO VS SPRING BOOT

- Go-based microservices in a Spring Cloud / Netflix OSS landscape requires more up-front work for integrating with the support components.
- With some basic software craftsmanship and code reuse I personally don't think it's a very big deal.
- I have published a Go-based microservice integration library for Spring Cloud / Netflix OSS on github:
  - https://github.com/eriklupander/cloudtoolkit
- And there are a number of other more or less general purpose microservice toolkits for Go:
  - go-kit, kite, micro, gizmo

ARCHITECTURAL OVERVIEW



Things not really Go-related…

EDGE SERVER

- Our Go services doesn't care about the EDGE / reverse-proxy
- Netflix Zuul, Nginx, HAProxy …
- Must forward HTTP headers.

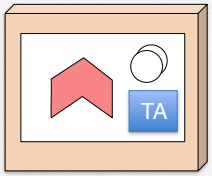
SERVICE DISCOVERY AND LOAD BALANCING

- Load-balancing and Service Discovery is handled by the orchestration engine.
  - E.g. the Docker Swarm or Kubernetes ”Service” abstraction.
- Eureka service discovery and Ribbon-like client-based load-balancing is easily implemented too.

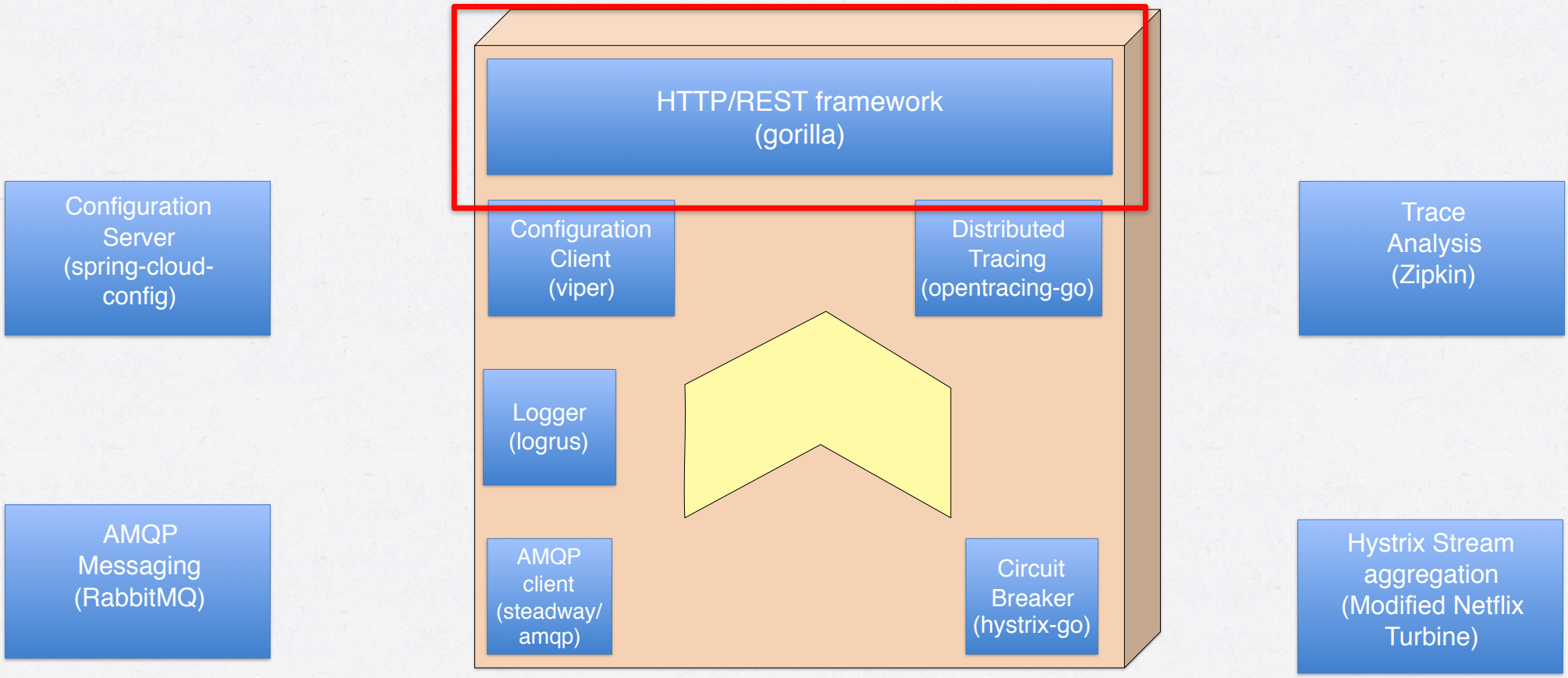
```
errors := hystrix.Go("get_account", func() error {
        req, _ := http.NewRequest("GET", "http://accountservice:7777/accounts/" + accountId, nil)
```

Demo 2 -
Load balancing and fast scaling
@ Docker Swarm

Go Microservice Anatomy



HTTP / REST FRAMEWORK



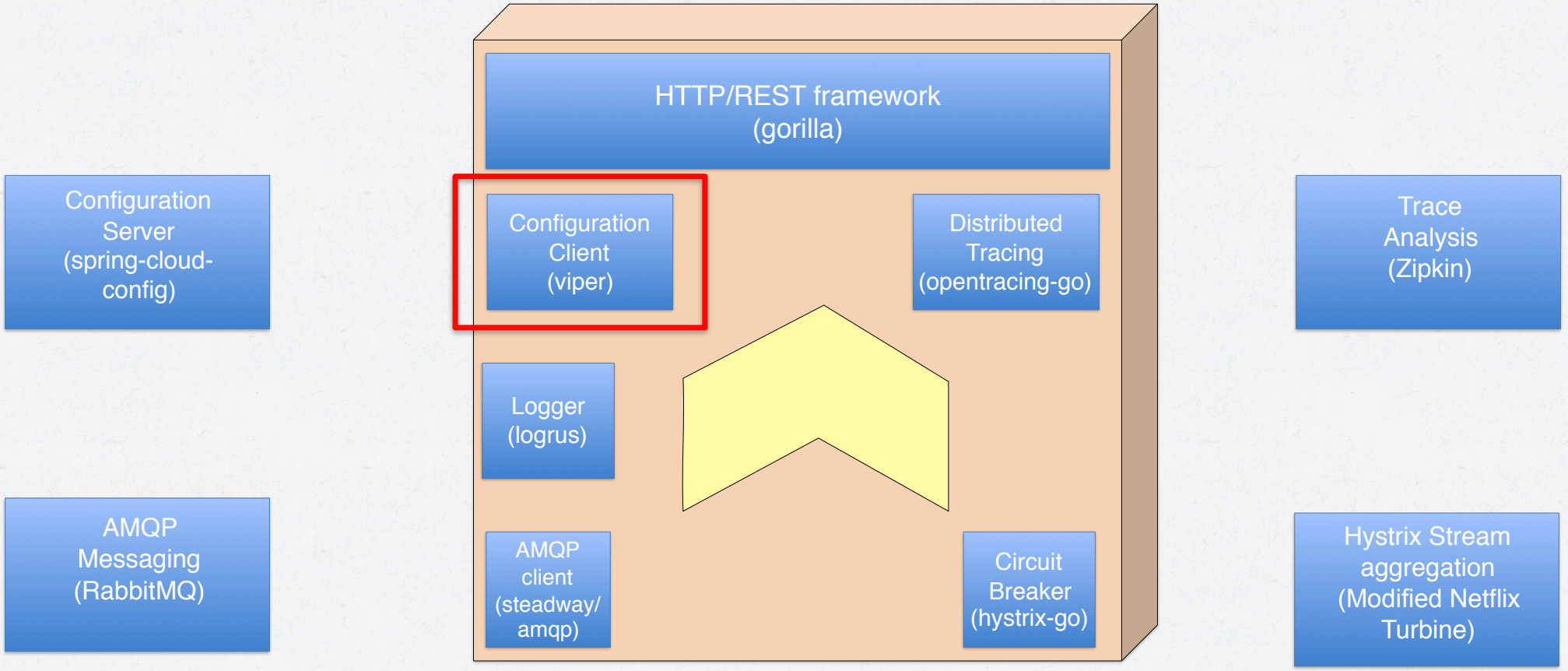
HTTP FRAMEWORK (GORILLA)

```
var routes = Routes{

        Route{
                "GetAccount",
                "GET",
                "/accounts/{accountId}",
                GetAccount,
        },
        Route{
                "HealthCheck",
                "GET",
                "/health",
                func(w http.ResponseWriter, r *http.Request) {
                        w.Header().Set("Content-Type", "text/plain; charset=UTF-8")
                        w.Write([]byte("OK"))
                },
        },
}
```

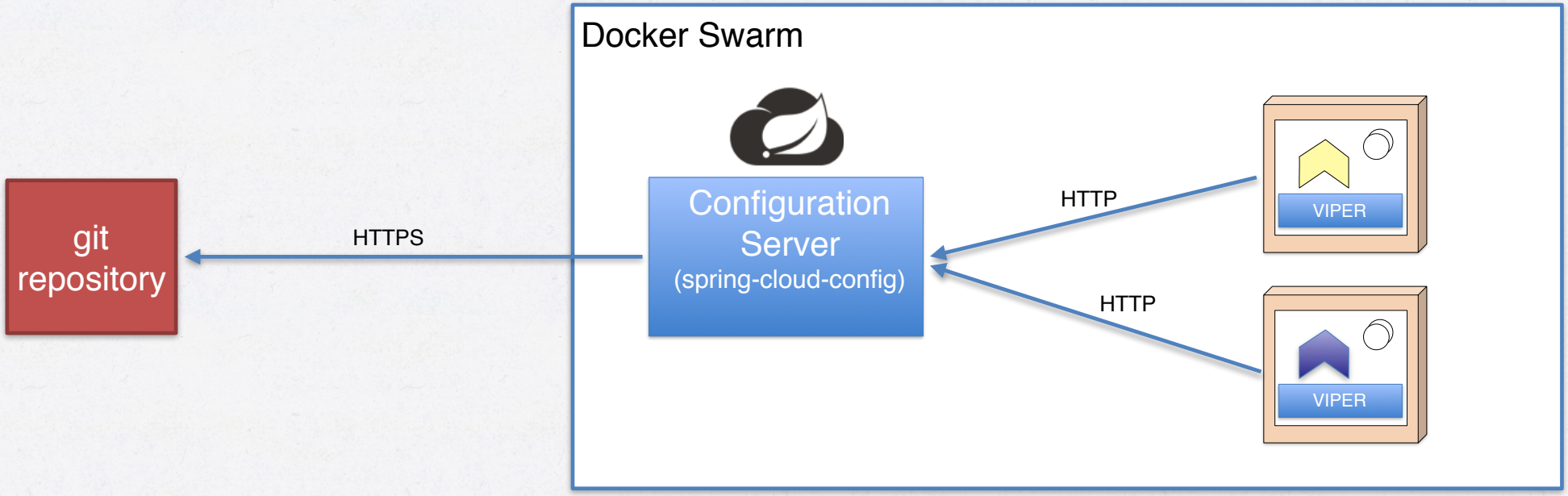
HTTP FRAMEWORK (GORILLA)

```
func GetAccount(w http.ResponseWriter, r *http.Request) {

        var accountId = mux.Vars(r)["accountId"]
        account, _ := client.GetAccount(accountId)
        data, _ := json.Marshal(account)

        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(http.StatusOK)
        w.Write(data)
}
```

CONFIGURATION



CONFIGURATION USING SPRING CLOUD CONFIG AND VIPER



CONFIGURATION - VIPER

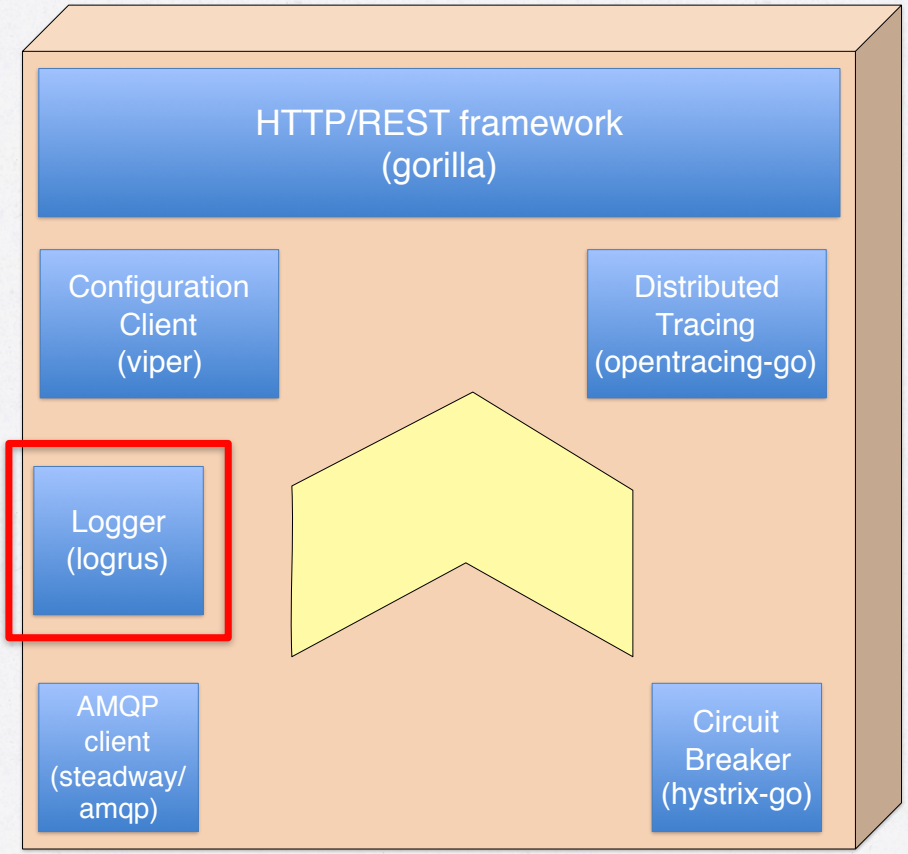
- Viper supports YAML, .properties, JSON and Env-vars
- With a few lines of code, we can load and inject config from Spring Cloud Config into Viper

```
resp, err := http.Get("http://configserver:8888/" + appName + "-" + envProfile + "/" + envProfile)
if err != nil {
        panic("Failed to load configuration: " + err.Error())
}

body, err := ioutil.ReadAll(resp.Body)

var cloudConfig model.SpringCloudConfig
json.Unmarshal(body, &cloudConfig)

for key, value := range cloudConfig.PropertySources[0].Source {
        viper.Set(key, value)
}
viper.SetConfigType("json")
```

CONFIGURATION - VIPER USAGE

```
go service.StartWebServer(viper.GetString("server_port")) // Starts HTTP service
```

LOGGING



Configuration Server (spring-cloud-config)

AMQP Messaging (RabbitMQ)

Trace Analysis (Zipkin)

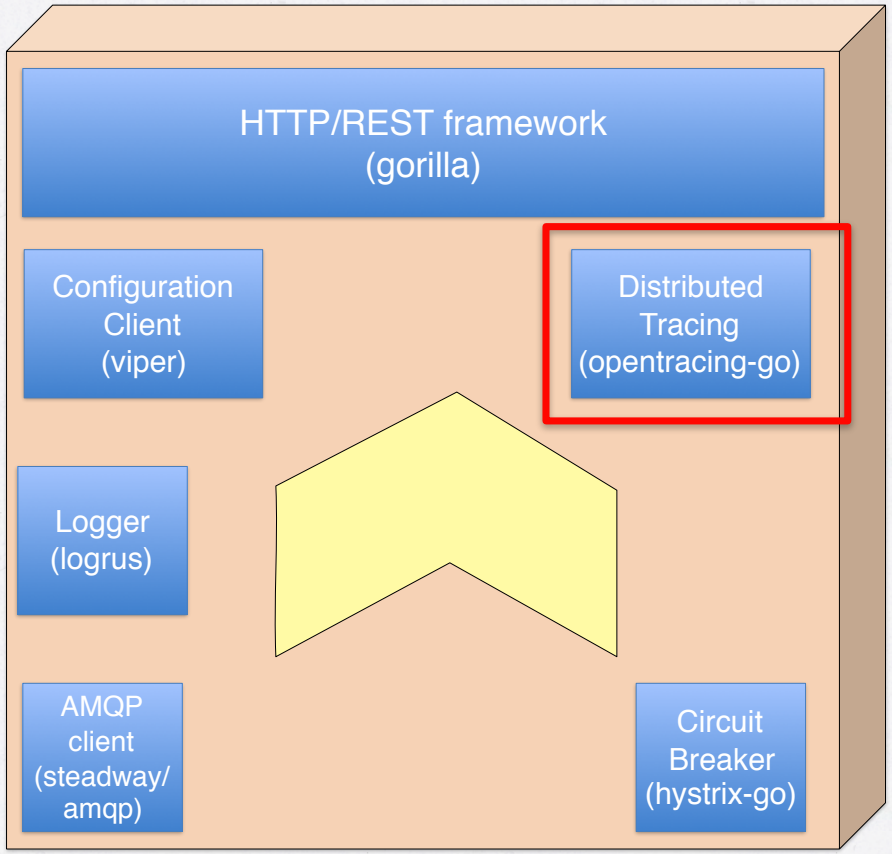
Hystrix Stream aggregation (Modified Netflix Turbine)

LOGGING - LOGRUS

- Application logs with 3rd party library Logrus
- Supports levels, fields, formatters
- 30+ built-in hooks

```
INFO[0000] A group of walrus emerges from the ocean      animal=walrus size=10
WARN[0000] The group's number increased tremendously!    number=122 omg=true
INFO[0000] A giant walrus appears!                       animal=walrus size=10
INFO[0000] Tremendously sized cow enters the ocean.      animal=walrus size=9
FATA[0000] The ice breaks!                               number=100 omg=true
exit status 1
```

DISTRIBUTED TRACING



Configuration Server (spring-cloud-config)

AMQP Messaging (RabbitMQ)

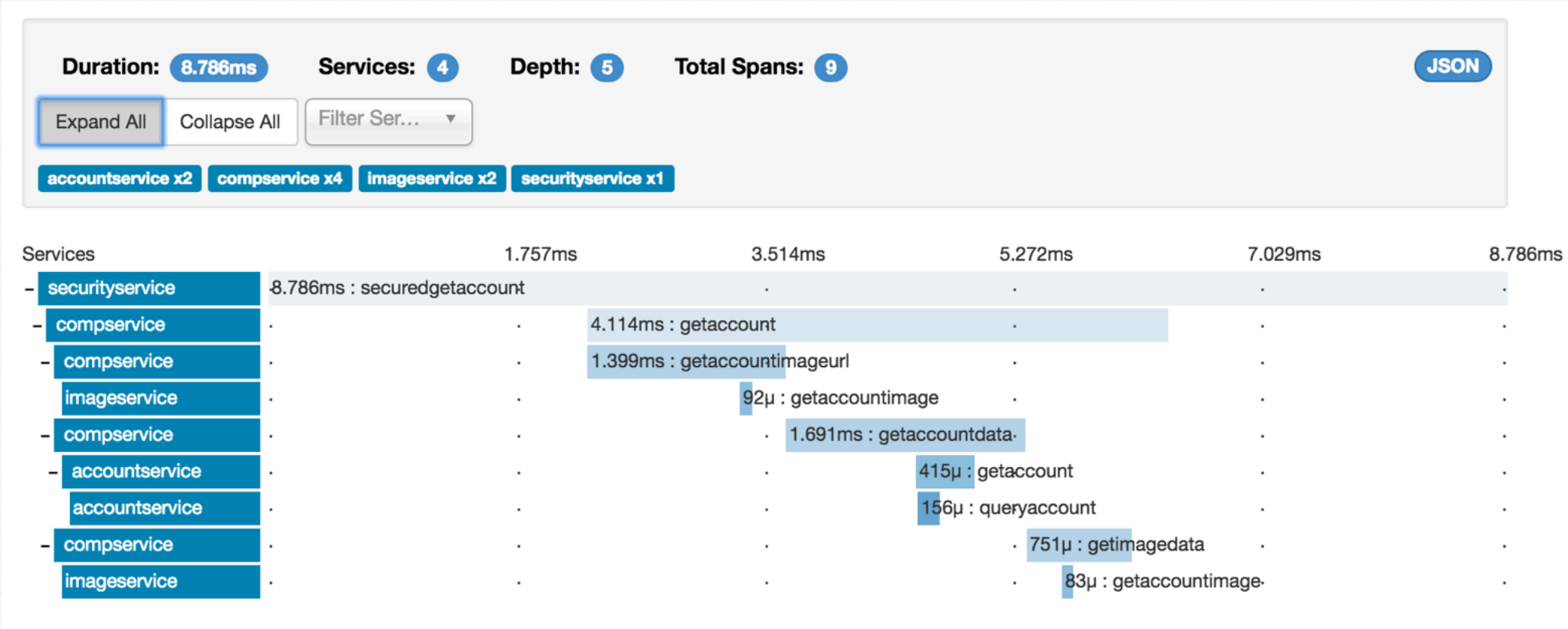
Trace Analysis (Zipkin)

Hystrix Stream aggregation (Modified Netflix Turbine)

DISTRIBUTED TRACING

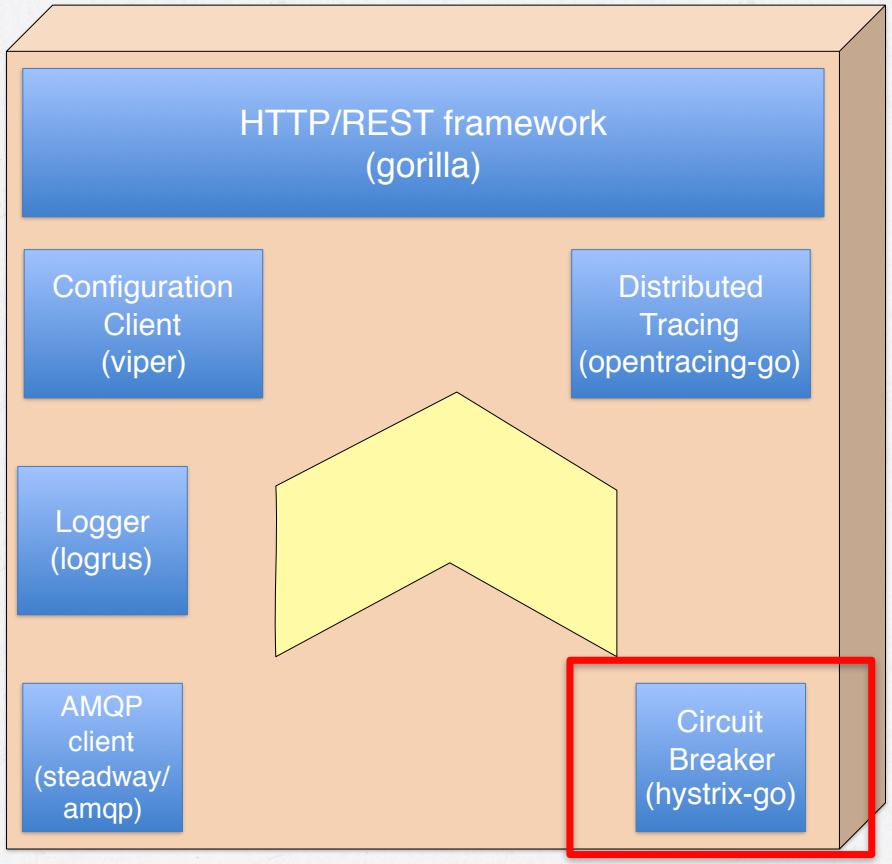
- Track a request over multiple microservices
  - Also trace within services and methods
- Invaluable for high-level profiling across the service stack.
- go-opentracing and zipkin

DISTRIBUTED TRACING - ZIPKIN



CALLISTA
— ENTERPRISE —

CIRCUIT BREAKER



Configuration Server (spring-cloud-config)

AMQP Messaging (RabbitMQ)

Trace Analysis (Zipkin)

Hystrix Stream aggregation (Modified Netflix Turbine)

CIRCUIT BREAKING - HYSTRIX

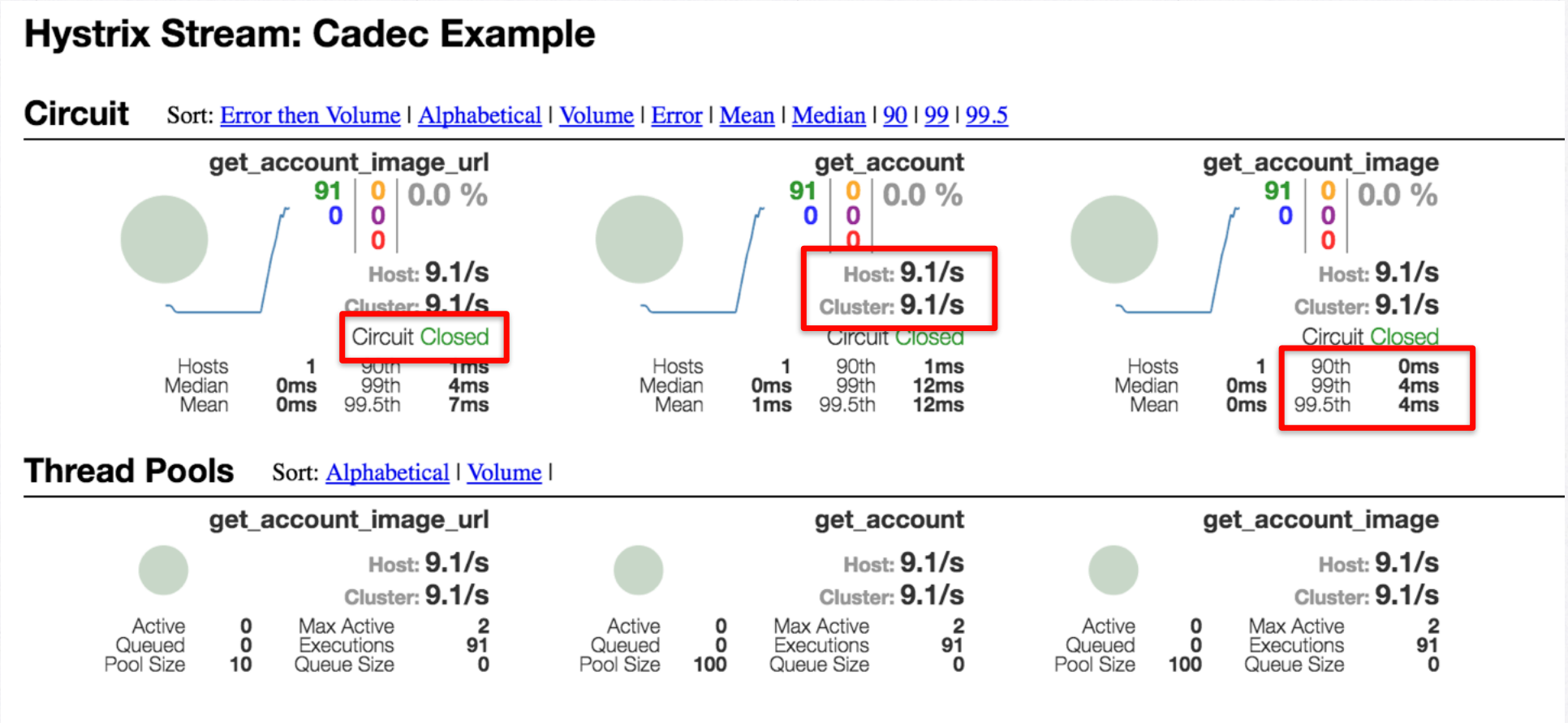
- Mechanism to make sure a single malfunctioning microservice doesn't halt the entire service or application.
- go-hystrix (circuit breaker)
- Netflix Turbine (aggregation)
- Netflix Hystrix (dashboard)

CIRCUIT BREAKING

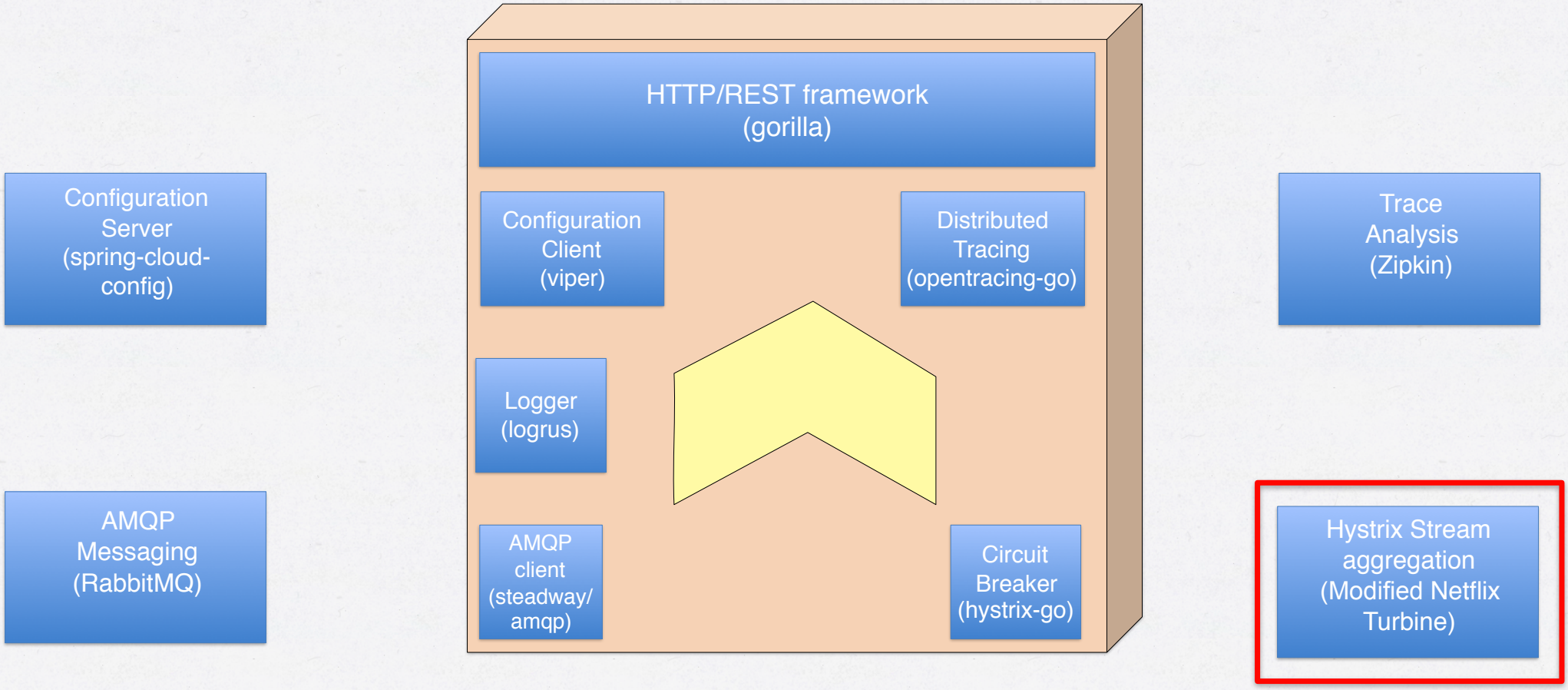
- Example go-hystrix usage, non-blocking.

```
output := make(chan []byte, 1)
errors := hystrix.Go("get_account", func() error {
        output <- getData(accountId)
        return nil
}, func(err error) error {
        // fallback method here
        return nil
})
```

CIRCUIT BREAKING - HYSTRIX DASHBOARD

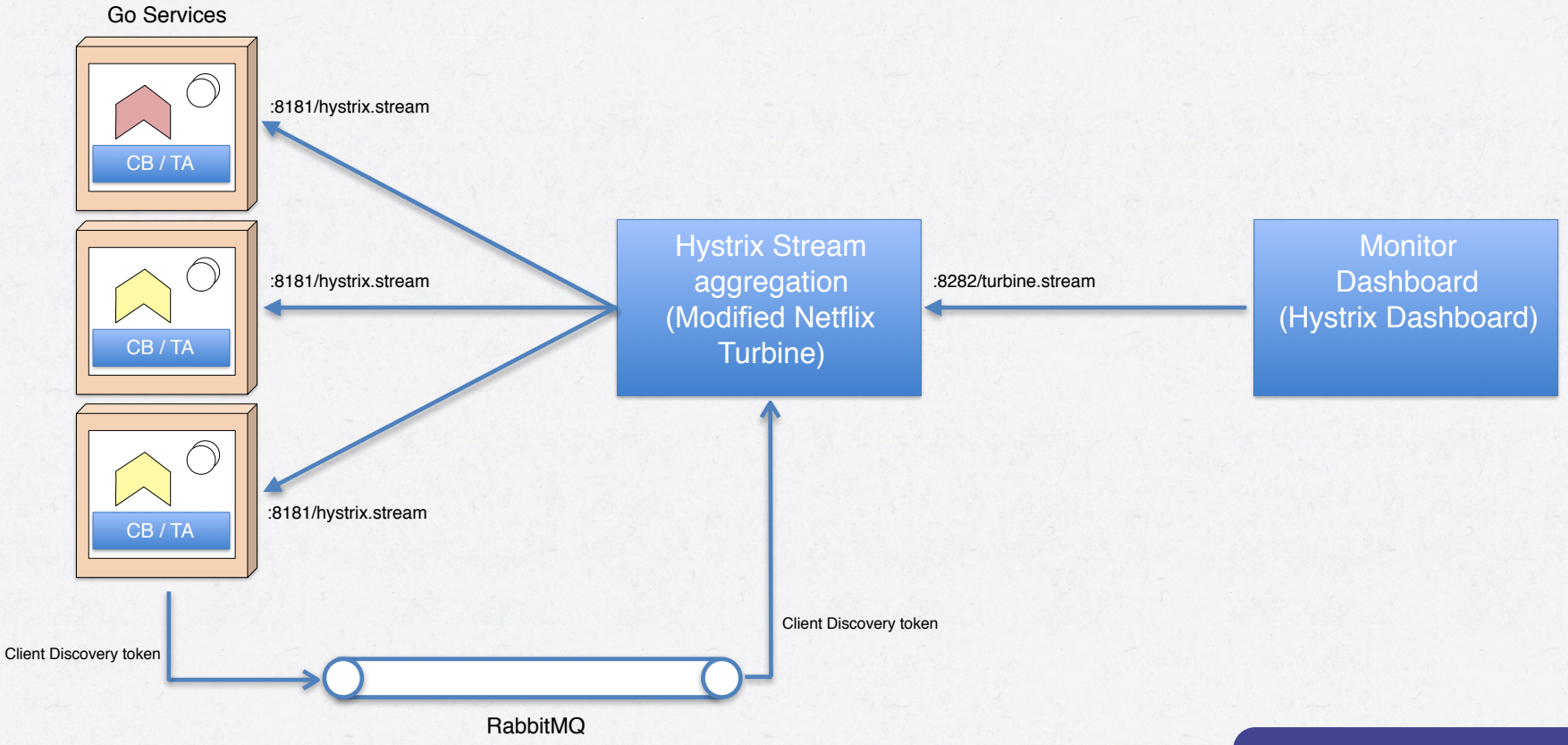


HYSTRIX STREAM AGGREGATION



CIRCUIT BREAKING

- Hystrix stream aggregation using customized Netflix Turbine



CALLISTA
– ENTERPRISE –

CIRCUIT BREAKING

- Programmatic hystrix configuration

```
func configureHystrix() {

        hystrix.ConfigureCommand("get_account_image", hystrix.CommandConfig{
                Timeout:                3000,
                MaxConcurrentRequests: 100,
                ErrorPercentThreshold: 25,
        })
        hystrix.ConfigureCommand("get_account", hystrix.CommandConfig{
                Timeout:                3000,
                MaxConcurrentRequests: 100,
                ErrorPercentThreshold: 25,
        })
```

SOME GENERAL CONSIDERATIONS

- Stability
  - Let it crash.
  - Let unrecoverable errors panic the microservice, let the container orchestrator handle restarts.
  - Use HEALTHCHECK for liveness.
- Security
  - EDGE + Auth Service
  - Security context passed down the microservice ”stack” using Go’s standard Context object and HTTP headers.
- Testing of microservices
  - Write unit tests as usual.
  - Leverage Docker and the Go Docker Remote API to build integration tests with dependencies that’s started with *go test.*
  - I strongly recommend looking into *net/http/httptest* and GoConvey

SAMPLE CODE 3 - TESTING WITH GOCONVEY

```
var sampleToken = "my-oauth-token"

func TestOAuthSpec(t *testing.T) {

        Convey("Given a HTTP request with a valid sample token", t, func() {
                httpReq := http.Request{
                        Header: map[string][]string{
                                "Authorization": {"Bearer: " + sampleToken},
                        },
                }

                Convey("When the auth header is extracted", func() {
                        token := extractAuthorizationFromHeader(&httpReq)

                        Convey("The auth header extracted should match the sample one", func() {
                                So(token, ShouldNotBeNil)
                                So(token, ShouldEqual, sampleToken)
                        })
                })
        })
}
```

SUMMARY

- Go is an interesting option for microservices due to runtime characteristics and rather pleasant developing.
- Genereally speaking, developing software in Go is often productive and quite fun, but not without it’s fair share of quirks especially regarding the lack of traditional OO constructs and missing generics.
- Microservice development in Go requires a bit of work regarding integration with supporting services, but can be mitigated by using integration libraries such as go-kit or our own little cloud-toolkit.
- Don’t be afraid to pick your favorites!

WANT TO LEARN MORE?

- Coming spring 2017 from Packt

- Technical reviewers:
  - Magnus Larsson
  - Erik Lupander

DVIZZ - A DOCKER SWARM VISUALIZER

- https://github.com/eriklupander/dvizz
- Pull requests are more than welcome!

RESOURCES

- Demo services: https://github.com/callistaenterprise/gocadec
- go-kit: https://github.com/go-kit/kit
- cloud-tookit: https://github.com/eriklupander/cloudtoolkit
- dvizz: https://github.com/eriklupander/dvizz
- packt book: https://www.packtpub.com/application-development/building-microservices-go

Questions?