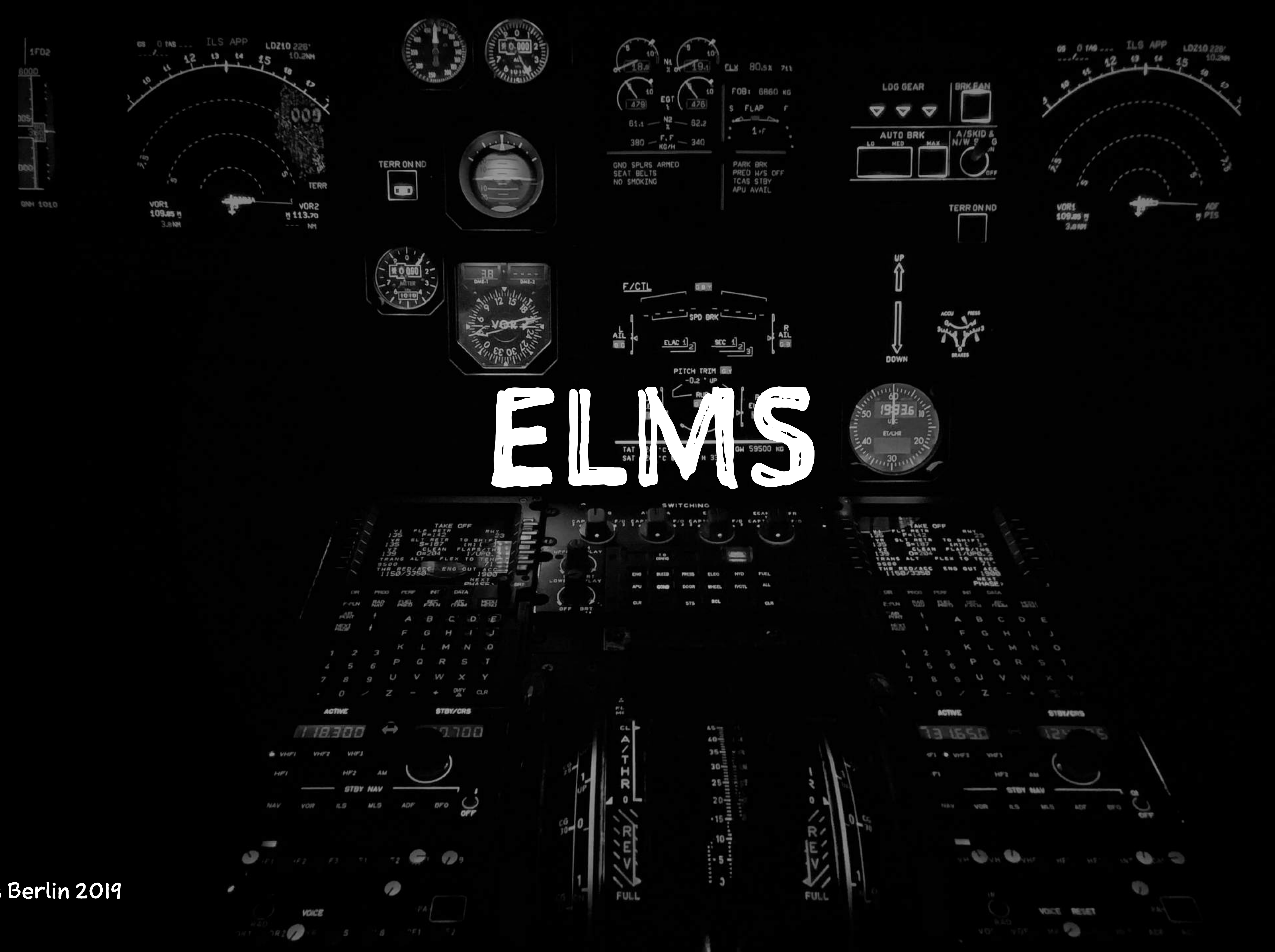
Designing for Failure

@italolelis

Go Days Berlin 2019

Think about an airplane, a boing 777 to be especific



ELMS

Go Days Berlin 2019

Essentials

Resilience is a Requirement, Not a Feature

Liang Guo

Dependency Isolation and Graceful Degradation

Health-check and Load Balancing

```go
import (
  "net/http"
  "time"

  "github.com/hellofresh/health-go"
  healthMysql "github.com/hellofresh/health-go/checks/mysql"
)

func main() {
  health.Register(health.Config{
    Name: "kafka",
    Timeout: time.Second*5,
    SkipOnErr: true,
    Check: func() error {
      // kafka health check implementation goes here
    },
  })

  health.Register(health.Config{
    Name:      "mysql",
    Timeout:   time.Second * 2,
    SkipOnErr: false,
    Check: healthMysql.New(healthMysql.Config{
      DSN: "test:test@tcp(0.0.0.0:31726)/test?charset=utf8",
    },
  })

  http.Handle("/status", health.Handler())
  http.ListenAndServe(":8080", nil)
}
```

If everything is OK you get...

```json
{
  "status": "OK",
  "timestamp": "2017-01-01T00:00:00.413567856+033:00",
  "system": {
    "version": "go1.8",
    "goroutines_count": 4,
    "total_alloc_bytes": 21321,
    "heap_objects_count": 21323,
    "alloc_bytes": 234523
  }
}
```

If things are not good but your app still can work...

```
{
  "status": "Partially Available",
  "timestamp": "2017-01-01T00:00:00.413567856+033:00",
  "failures": {
    "rabbitmq": "Failed during rabbitmq health check"
  },
  "system": {
    "version": "go1.8",
    "goroutines_count": 4,
    "total_alloc_bytes": 21321,
    "heap_objects_count": 21323,
    "alloc_bytes": 234523
  }
}
```

Otherwise...

```
{
  "status": "Unavailable",
  "timestamp": "2017-01-01T00:00:00.413567856+033:00",
  "failures": {
    "mongodb": "Failed during mongodb health check"
  },
  "system": {
    "version": "go1.8",
    "goroutines_count": 4,
    "total_alloc_bytes": 21321,
    "heap_objects_count": 21323,
    "alloc_bytes": 234523
  }
}
```

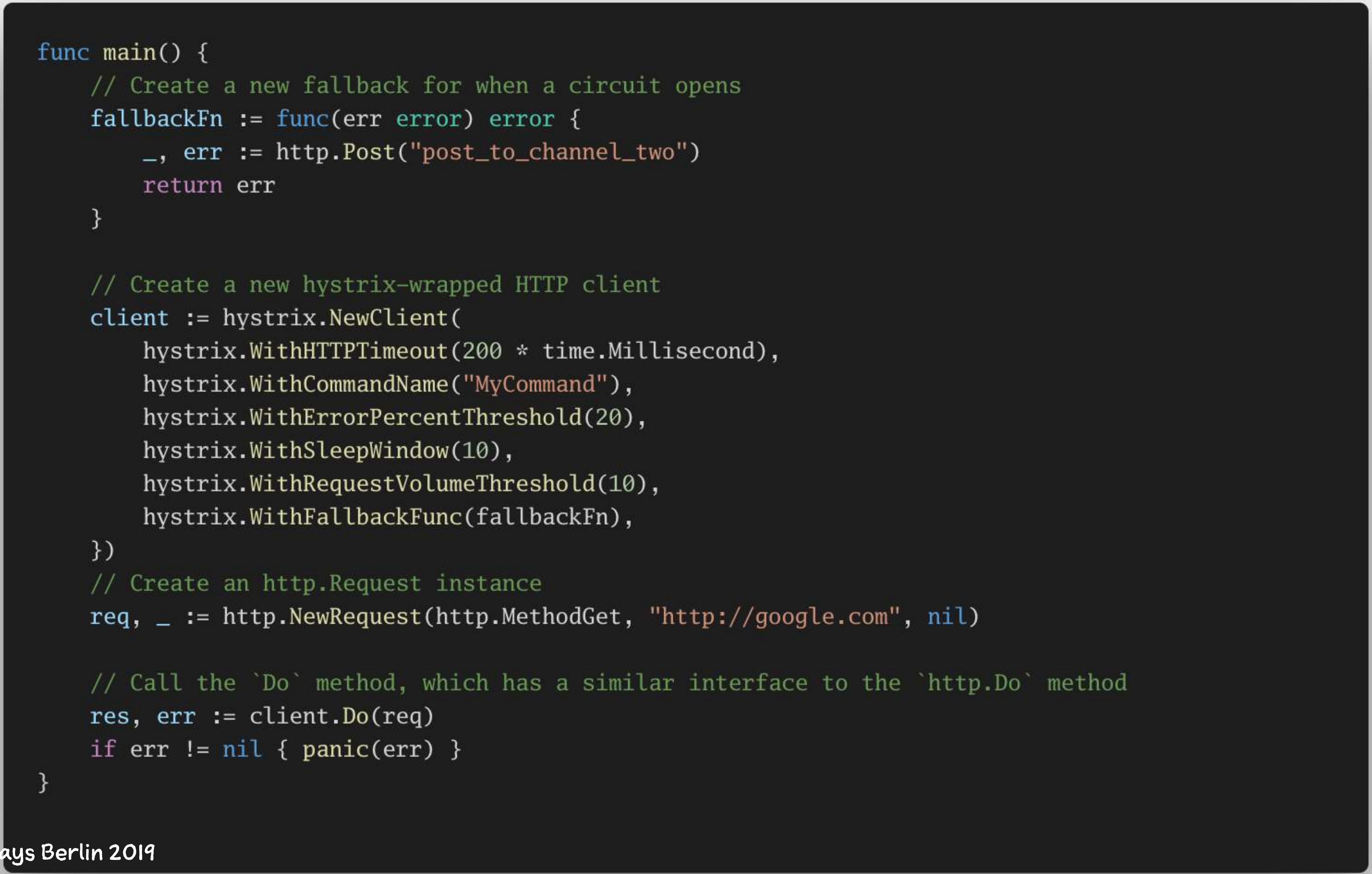
Self-healing

In kube this is as simple as defining a YAML file rule

```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: coffee-shop-deploy
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: coffee-shop
    spec:
      containers:
      - name: coffee-shop
        image: italolelis/coffee-shop:0.5.0
        ports:
        - containerPort: 9876
        env:
        - name: VERSION
          value: "0.9"
```

Load shedding

Circuit Breakers

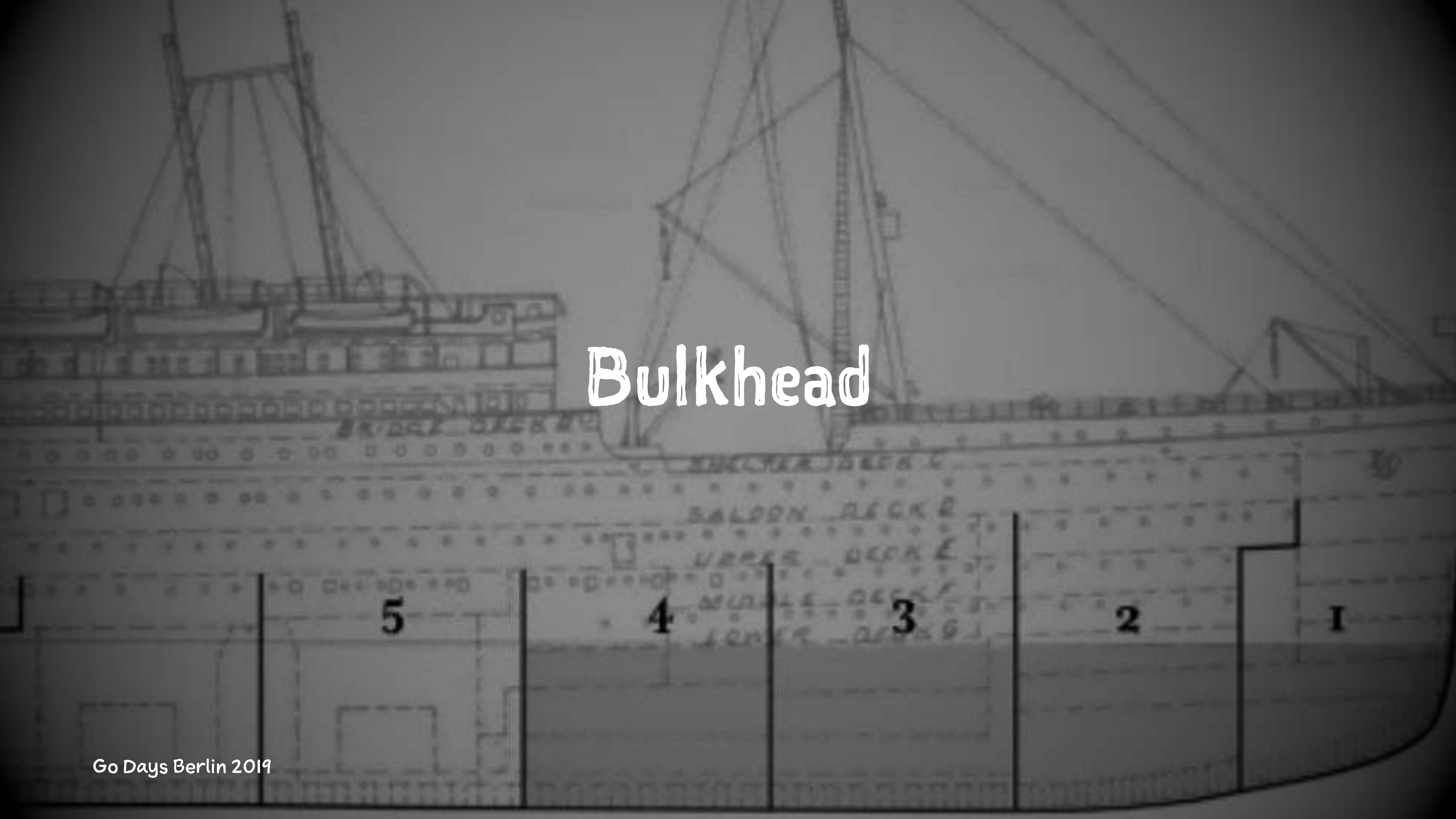


```go
func main() {
    // Create a new fallback for when a circuit opens
    fallbackFn := func(err error) error {
        _, err := http.Post("post_to_channel_two")
        return err
    }

    // Create a new hystrix-wrapped HTTP client
    client := hystrix.NewClient(
        hystrix.WithHTTPTimeout(200 * time.Millisecond),
        hystrix.WithCommandName("MyCommand"),
        hystrix.WithErrorPercentThreshold(20),
        hystrix.WithSleepWindow(10),
        hystrix.WithRequestVolumeThreshold(10),
        hystrix.WithFallbackFunc(fallbackFn),
    })
    // Create an http.Request instance
    req, _ := http.NewRequest(http.MethodGet, "http://google.com", nil)

    // Call the `Do` method, which has a similar interface to the `http.Do` method
    res, err := client.Do(req)
    if err != nil { panic(err) }
}
```

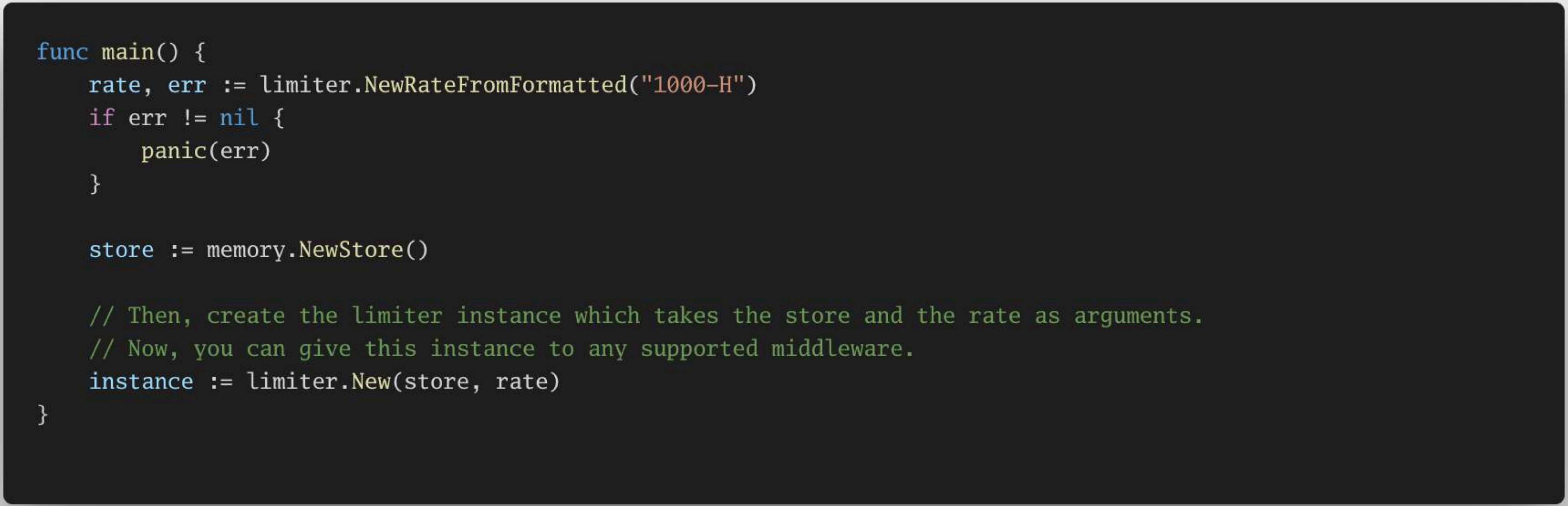
Retry Logic

```go
func main() {
    // Exponential Backoff increases the backoff at a exponential rate
    initTimeout := 2*time.Millisecond
    maxTimeout := 10*time.Millisecond
    expFactor := 2
    maxJitterInterval := 2*time.Millisecond

    backoff := heimdall.NewExponentialBackoff(
        initTimeout,
        maxTimeout,
        expFactor,
        maxJitterInterval,
    )

    // Create a new retry mechanism with the backoff
    retrier := heimdall.NewRetrier(backoff)


// Create a new client, set the retry mechanism, and the number of times you would like to retry
    client := httpclient.NewClient(
        httpclient.WithHTTPTimeout(1000 * time.Millisecond),
        httpclient.WithRetrier(retrier),
        httpclient.WithRetryCount(4),
    )
    // Create an http.Request instance
    req, _ := http.NewRequest(http.MethodGet, "http://google.com", nil)

    // Call the `Do` method, which has a similar interface to the `http.Do` method
    res, err := client.Do(req)
    if err != nil { panic(err) }
}
```

Bulkhead


BRIDGE DECK
SHELTER DECK C
SALOON DECK D
UPPER DECK E
MIDDLE DECK F
LOWER DECK G
5
4
3
2
1

Rate Limiters



```go
func main() {
    rate, err := limiter.NewRateFromFormatted("1000-H")
    if err != nil {
        panic(err)
    }

    store := memory.NewStore()

    // Then, create the limiter instance which takes the store and the rate as arguments.
    // Now, you can give this instance to any supported middleware.
    instance := limiter.New(store, rate)
}
```

Outbox Pattern

Outlier Server Host Detection

Service Mesh

SLO's and SLI's

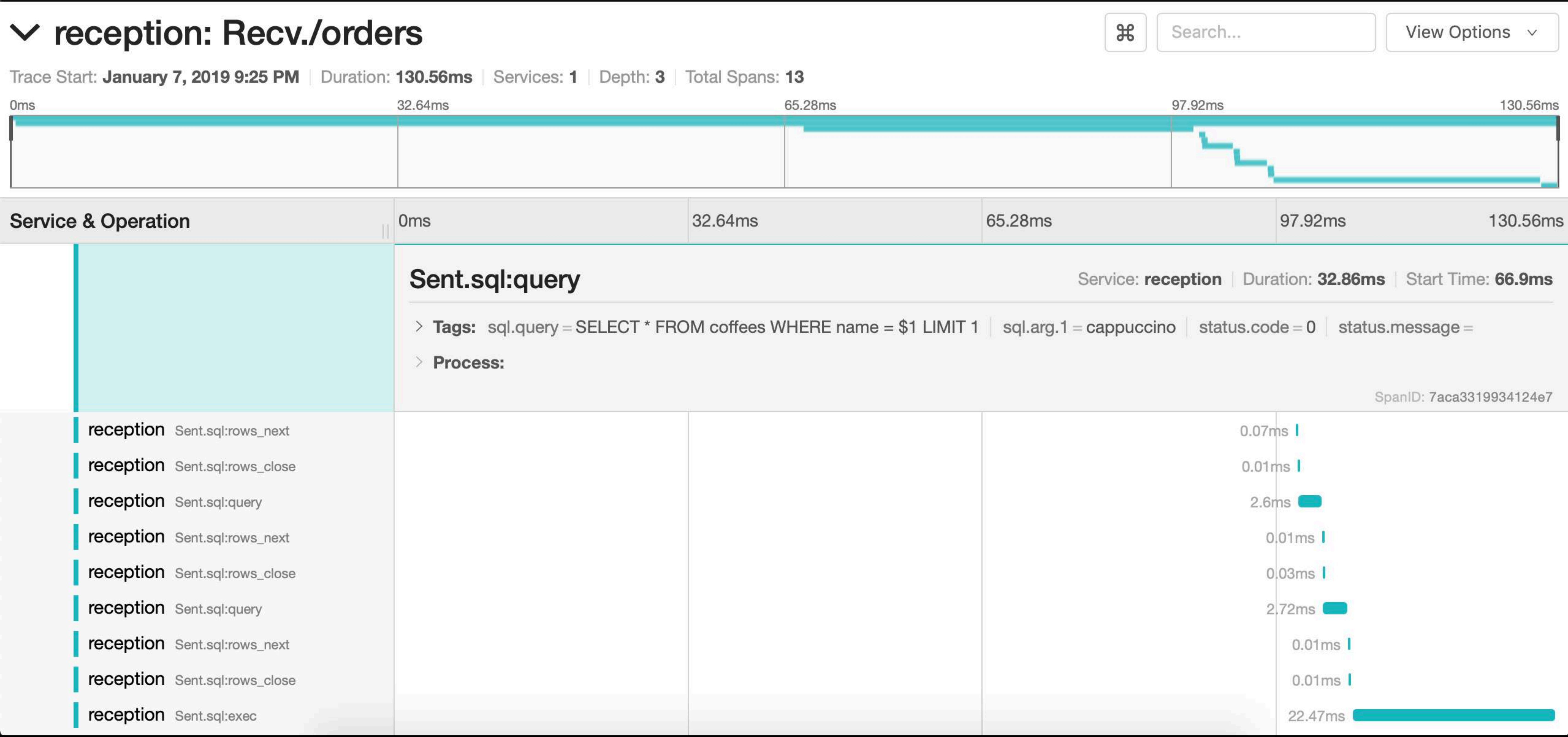
Dashboard
Zoom Out
Oct 4, 2015 11:29:09 to Oct 4, 2015
Logins
172
Sign ups
263
Sign outs
268
Support calls
80
Memory / CPU
logins
server requests
upper_25
upper_50
upper_75
upper_90
upper_95
web_server_01
web_server_02
web_server_03
web_server_04

Monitoring

```go
if err := view.Register(
        ochttp.ClientSentBytesDistribution,
        ochttp.ClientReceivedBytesDistribution,
        ochttp.ClientRoundtripLatencyDistribution,
    ); err != nil {
        logger.Fatal(err)
    }

    exporter, err := prometheus.NewExporter(prometheus.Options{
        Namespace: cfg.ServiceName,
    })
    if err != nil {
        log.Fatal("failed to create the prometheus stats exporter")
    }
    view.RegisterExporter(exporter)
    view.SetReportingPeriod(cfg.ReportingPeriod)
```

Distributed Tracing

```
exporter, err := jaeger.NewExporter(jaeger.Options{
    CollectorEndpoint: cfg.CollectorEndpoint,
    Process: jaeger.Process{
        ServiceName: cfg.ServiceName,
    },
})
if err != nil {
    log.Error("could not create the jaeger exporter")
}

trace.RegisterExporter(exporter)
trace.ApplyConfig(trace.Config{DefaultSampler: trace.AlwaysSample()})
```



reception: Recv./orders

Search... | View Options

Trace Start: **January 7, 2019 9:25 PM** | Duration: **130.56ms** | Services: **1** | Depth: **3** | Total Spans: **13**

0ms 32.64ms 65.28ms 97.92ms 130.56ms

Service & Operation | 0ms | 32.64ms | 65.28ms | 97.92ms | 130.56ms

Sent.sql:query

Service: **reception** | Duration: **32.86ms** | Start Time: **66.9ms**

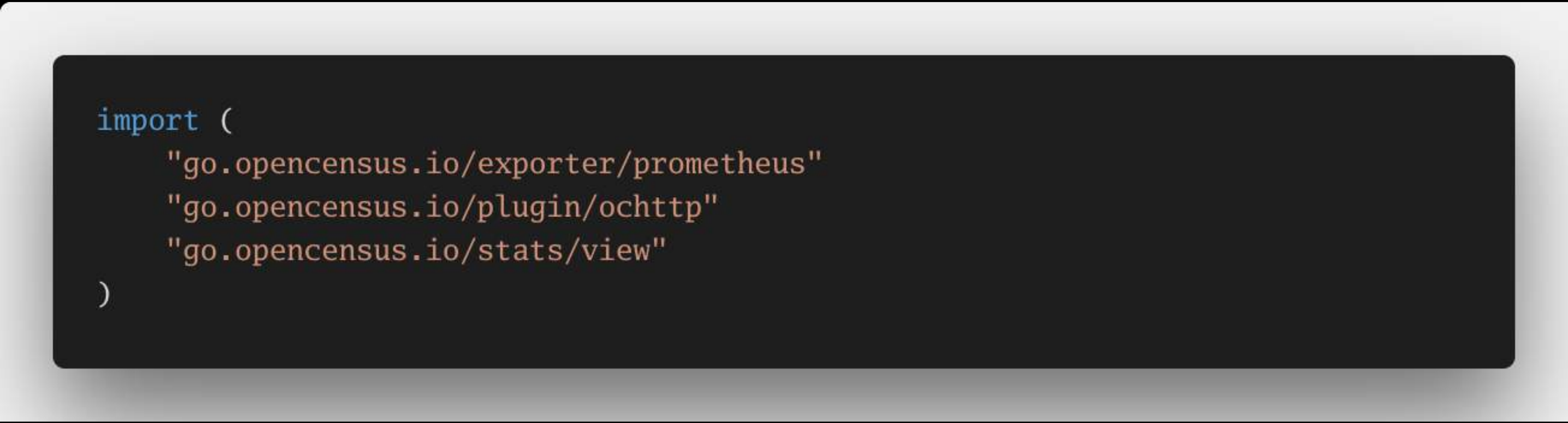
Tags: sql.query = SELECT * FROM coffees WHERE name = $1 LIMIT 1 | sql.arg.1 = cappuccino | status.code = 0 | status.message =

Process:

SpanID: 7aca3319934124e7

reception	Sent.sql:rows_next	0.07ms
reception	Sent.sql:rows_close	0.01ms
reception	Sent.sql:query	2.6ms
reception	Sent.sql:rows_next	0.01ms
reception	Sent.sql:rows_close	0.03ms
reception	Sent.sql:query	2.72ms
reception	Sent.sql:rows_next	0.01ms
reception	Sent.sql:rows_close	0.01ms
reception	Sent.sql:exec	22.47ms

Open Census



```go
import (
    "go.opencensus.io/exporter/prometheus"
    "go.opencensus.io/plugin/ochttp"
    "go.opencensus.io/stats/view"
)
```

Recap

1. Always think about your dependencies
2. Dependency Isolation and Graceful Degradation
3. Load shedding and Request Controlling
4. Observalibility is not optional

Questions and links!

- Example application: https://github.com/italolelis/coffee-shop
- Link to the slides: https://github.com/italolelis/talks

Thank you!

Go Days Berlin 2019