Building cloud native service in Go

GopherCon 2021 Workshop

Eran Levy
@levyeran

Introduction

Eran Levy

Cyren, 2.8y~ ago we've done the transition to Go

levyeran.medium.com

@levyeran

Quick poll

https://www.surveymonkey.co.uk/r/6M6V755

WIFM?

We will write (end-to-end) a new Go service designed for the cloud

We will focus on the libraries, tools and best practices to build such service

This workshop assumes you have basic Go knowledge

Agenda

- Fundamentals
- Build HTTP APIs - REST
- Build HTTP APIs - gRPC
- Logging
- Metrics & Tracing with OpenTelemetry
- Caching
- Database persistence
- Containerization & Deployment

So what is this buzzword - “Cloud Native” ?

Cloud native technologies empower organizations to build and run **scalable** applications in modern, **dynamic** environments such as public, private, and hybrid clouds.

Containers, service meshes, microservices, immutable infrastructure, and declarative APIs **exemplify** this approach.

These techniques enable **loosely coupled** systems that are **resilient, manageable, and observable**. Combined with robust automation, they allow **engineers to make high-impact changes** frequently and predictably with minimal toil.

https://github.com/cncf/toc/blob/master/DEFINITION.md

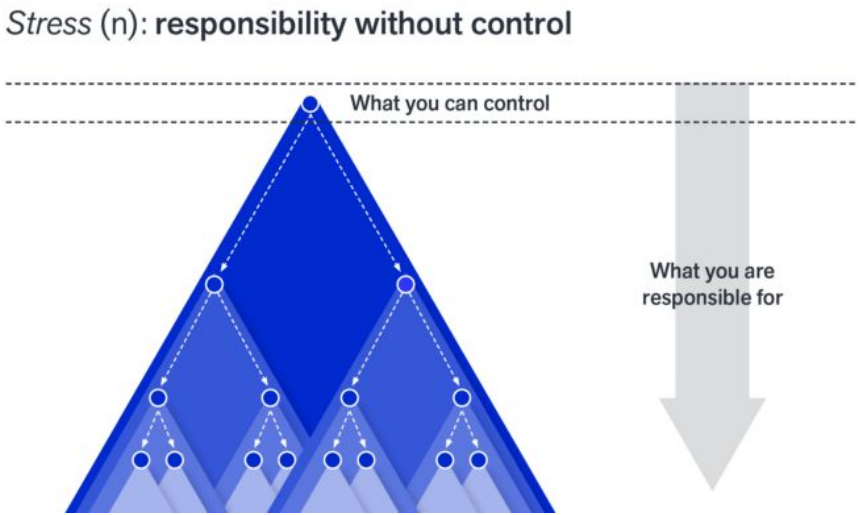
You might be interested in reading my post, what it means from an engineer perspective -

“The Cloud Native Engineer: The engineer evolution at a glance”

“Deep systems”

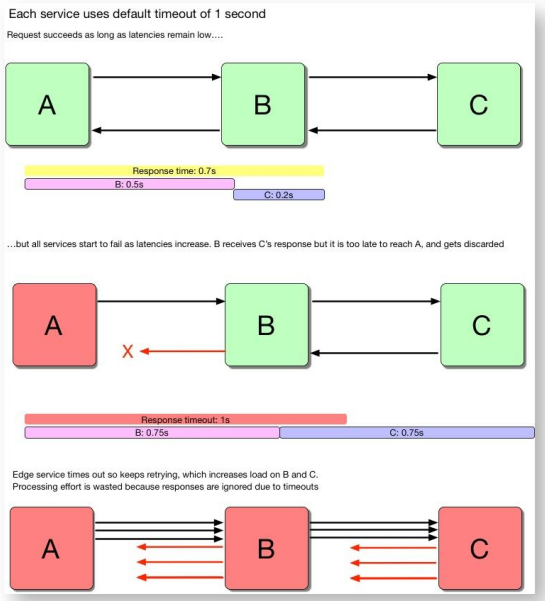
As engineers we usually ship

part of a larger piece of software



Source: https://lightstep.com/deep-systems/

A real world example



https://engineering.grab.com/context-deadlines-and-how-to-set-them

Talking about Cloud Native without mentioning...

Fallacies of distributed computing:

1. The network is reliable;
2. Latency is zero;
3. Bandwidth is infinite;
4. The network is secure;
5. Topology doesn't change;
6. There is one administrator;
7. Transport cost is zero;
8. The network is homogeneous.
9. We all trust each other.

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing#The_fallacies

Dynamic, scalable, resilient, observable...

These environments are where **Go shines** 

Fast builds, strong standard libraries, low Memory/CPU footprint (--> cost?), goroutines, programming language (i.e. context.Context) ...

Enough with theory

Go project setup - Text tokenizer

- Clone https://github.com/eran-levy/gophercon-workshop to tokenizer-gophercon
- Git clone git@github.com:eran-levy/gophercon-workshop.git tokenizer-gophercon
- go mod init, run “hello world”

```
module github.com/eran-levy/tokenizer-gophercon

go 1.15
```

Project layout

- There is no one “rule them all” guideline
- https://github.com/golang-standards/project-layout
- There is some debt in the community around **/pkg** and **/internal** folders structure - I suggest you to read this post: https://travisjeffery.com/b/2019/11/i-ll-take-pkg-over-internal/
- Use /**cmd** to place different project executors (i.e. Server, CLI)
- Try to keep the root folder clean of Go files (tends to include Dockerfile, etc.)

Conventions will make it easier for new engineers to onboard easily, navigate projects that not familiar with...

Recommended talk - GopherCon 2018: Kat Zien - How Do You Structure Your Go Apps

Cloud Native services tend to be part of...

A larger piece of software. You project will evolve. You want to keep it simple from day one, but

- Cloud Native services tend to have multiple **inbound** channels (gRPC, REST, Kinesis, others)
- Cloud Native services tend to have multiple **outbound** channels (Data store, Kafka, Kinesis, others)
- Cloud Native services might interact with **other services** to get data

You need to structure your code in a way that will enable you to:

perform **changes easily** where possible and write **reliable tests** fast.

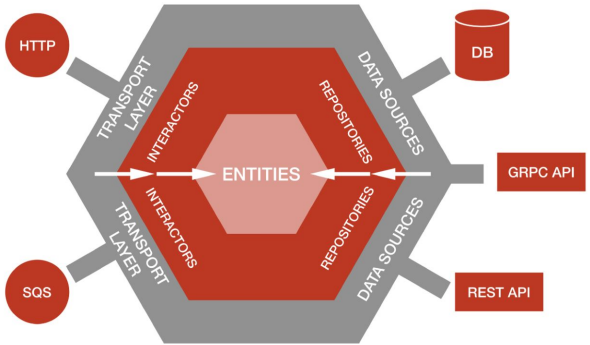
Choose a pattern

There are different ways to approach that - Clean Architecture, Hexagonal Architecture and more.

Hexagonal Architecture is my preferred pattern

- **Convention** to structure your codebase
- **Isolate** your core logic from external sources
- **Test** logics independently (i.e. contract testing)

Ports & Adapters - its all about making sure your core business logic is isolated from external dependencies using interfaces and transformers



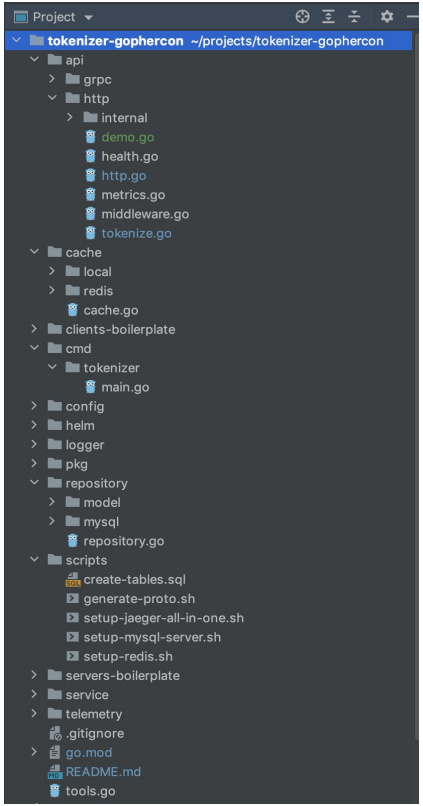
Source: netflixtechblog.com

How does it look like?

```go
package internal

type TokenizeTextAPIRequest struct {
    GlobalTxId string `json:"global_tx_id" binding:"required"`
    Txt        string `json:"text" binding:"required"`
}

type TokenizeTextAPIResponse struct {
    GlobalTxId   string   `json:"global_tx_id"`
    RequestId    string   `json:"request_id"`
    TokenizedTxt []string `json:"tokenized_text,omitempty"`
    NumOfWords   int      `json:"num_of_words"`
}
```

```go
package service

type TokenizeTextRequest struct {
    RequestId  string `json:"request_id"`
    GlobalTxId string `json:"global_tx_id"`
    Txt        string `json:"txt"`
}

type TokenizeTextResponse struct {
    RequestId    string   `json:"request_id"`
    TokenizedTxt []string `json:"tokenized_text,omitempty"`
    NumOfWords   int      `json:"num_of_words"`
}
```

```go
type TokenizerService interface {
    TokenizeText(ctx context.Context, request TokenizeTextRequest) (TokenizeTextResponse, error)
```



Project layout sample

Our application configuration

- While building a cloud native service, the 12 factor app methodology will help you to make sure you are on the right path
- Configuration varies between environments
- Configuration defaults
- Fail fast on required configurations
- Configurations are set on
    - Environment variables (i.e. injected with helm, secrets can also be injected) and
    - Config files (i.e k8s ConfigMap)

Go configuration

- Main will glue the configuration together with our app
- The injected env vars / files are parsed into the relevant parameter properties that can reside either in the config/ folder or in main itself (depends on the types of configs)
    - You can use “os”, spf13/viper, go-micro, ardanlabs/kit or any other to parse environment variables
    - Parsing the files depends mainly on the file type (yaml/json/key-val/etc) - you can use spf13/viper, go-micro, ardanlabs/kit, go yaml or anything else

Start and exit intelligently

- Start **fast** - in cloud native environments you scale up/down, you deploy whenever you need to, etc.
- You might be able to warm up on **first calls**
- Orchestrators have **startup probes** such as /health, /readiness that will enable you to utilize their mechanisms to react for situations (i.e. panic, disabled /health, etc)
- **Shutdown gracefully** with SIGTERM/SIGINT/others in order to complete your work intelligently and start serve in other replicas of your service (i.e. shutdown the web server, unsubscribe your kafka consumer, etc)

```
signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)
```

context.Context is a key

- Create context early (i.e. inbound request arrives)
- Context “carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.” (ref https://golang.org/pkg/context/)
- Context can carry request scoped key-value pairs for your logs (i.e. log your user id, transaction id, etc). OpenTelemetry is using Context to carry SpanContext
- Cancel parent context will cancel its child contexts
- Context can help you in terminating operations (i.e. goroutines, child perations, etc)

Before we move forward

Let’s write our service business logic API and model

Quick break

Bootstrap our HTTP REST API endpoint

- net/http standard library is rich and strong (https://golang.org/pkg/net/http/)
- The standard library arrives with ServeMux which probably good for many use cases
- There are libraries that extend the default multiplexer with url patterns, middlewares, and more. Good examples are: github.com/julienschmidt/httprouter and gorilla/mux (https://github.com/julienschmidt/go-http-routing-benchmark)
- Regular expression pattern matching in routers are dangerous
- gin-gonic/gin is taking a broader approach - defining itself as a framework which also ranked in its performance
- micro/micro , go-kit/kit are taking a much broader approach - microservice framework

Validation, error handling and response

Validate request

Provide meaningful error and response codes - clients shall know how to react to errors (bad request, server error, rate limit to try later, etc)

These are part of **your API contract**

Use middlewares

Move out any non-handler logic to middlewares

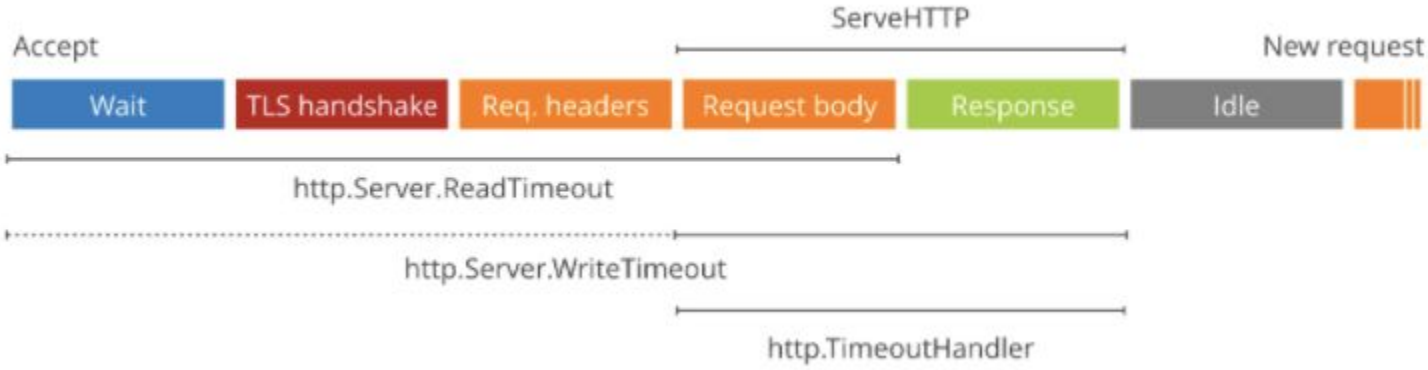
Middlewares can help you to apply same logic to different handlers

Service health

If your service is running in an orchestrator, you can utilize the health probes in order to “say” that the service isn’t healthy to get more traffic

For example: k8s liveness and readiness probes
https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes

HTTP server timeouts



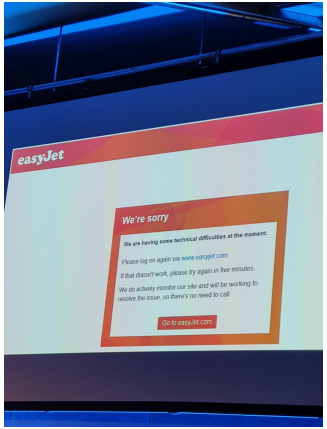
...very slow or disappearing clients might leak file descriptors...

```
srv := &http.Server{
    Addr:         s.cfg.HttpAddress,
    ReadTimeout:  s.cfg.ReadRequestTimeout,
    WriteTimeout: s.cfg.WriteResponseTimeout,
    Handler:     http.TimeoutHandler(r,2*time.Second, msg: ""),
}
s.srv = srv
if err := srv.ListenAndServe(); err != nil && errors.Is(err, http.ErrServerClosed) {
    logger.Log.Infof("http server closed #{err}")
    fatalErrors <- err
}
```

Source: https://blog.cloudflare.com/the-complete-guide-to-golang-net-http-timeouts/

Design with resilience in mind

- Embrace failures - they happen!
- Your client won't always do what you expect them to do… protect yourself (timeouts?)
- Handle errors gracefully - retry with backoff?
- Make sure your service startup is fast - sometimes you might need some help from the orchestrator you are running in
- Configure the context timeout to be the **sum of downstream timeouts including retries**



Source: QCon - Bernd Ruecker

Time to write our API adapter and port it to service

Protect our services - not just security!

Cloud native services are usually part of the puzzle - what does it mean to protect?

1. Rate limiting - how many requests our service can handle concurrently? Is it per customer, global, etc?
2. Sizing - can we receive request payloads greater than X MB?
3. Request handling timeouts (using context, connection timeouts, etc)
4. Circuit breakers (https://docs.microsoft.com/en-us/previous-versions/msp-n-p/dn589784(v=pandp.10))
5. …

You don't need to take care of everything -

API Gateway (?) might handle the rate limiting for you, **Envoy (?)** might take care of your retries, **service mesh (Istio?)** might taking care of the non-business logic boilerplate for you already...

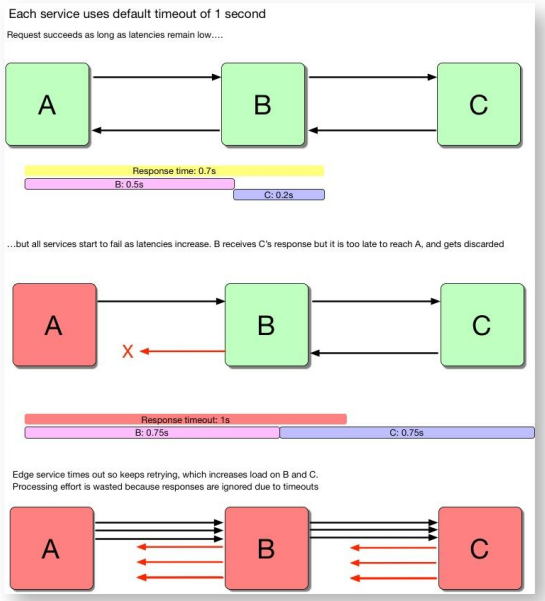
Context deadlines

Variables
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, d time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
type CancelFunc
type Context
  func Background() Context
  func TODO() Context
  func WithValue(parent Context, key, val interface{}) Context

```
package main

import (
        "context"
        "fmt"
        "time"
)

const shortDuration = 1 * time.Millisecond

func main() {
        // Pass a context with a timeout to tell a blocking function that it
        // should abandon its work after the timeout elapses.
        ctx, cancel := context.WithTimeout(context.Background(), shortDuration)
        defer cancel()

        select {
        case <-time.After(1 * time.Second):
                fmt.Println("overslept")
        case <-ctx.Done():
                fmt.Println(ctx.Err()) // prints "context deadline exceeded"
        }

}
```

Source: https://golang.org/pkg/context/#example_WithTimeout

Context deadlines



https://engineering.grab.com/context-deadlines-and-how-to-set-them

Context deadlines in practice

```
return internal.WithSpan(ctx, name: "redis.with_conn", func(ctx context.Context, span trace.Span) error {
    cn, err := c.getConn(ctx)
    if err ≠ nil : err ⤴

    if span.IsRecording() {
        if remoteAddr := cn.RemoteAddr(); remoteAddr ≠ nil {
            span.SetAttributes(label.String( k: "net.peer.ip", remoteAddr.String()))
        }
    }

    defer func() {
        c.releaseConn(ctx, cn, err)
    }()

    done := ctx.Done()
    if done == nil {
        err = fn(ctx, cn)
        return err
    }

    errc := make(chan error, 1)
    go func() { errc ← fn(ctx, cn) }()

    select {
    case ←done:
        _ = cn.Close()
        // Wait for the goroutine to finish and send something.
        ←errc

        err = ctx.Err()
        return err
    case err = ←errc:
        return err
    }
})
```

Source: go-redis - github.com/go-redis/redis/v8@v8.4.11/redis.go

Keep out transactions that will waste resources

- Set context timeouts in your inbound channels - you can't rely on somebody else to take care of that
- Use your context - libraries will help you to take care of the context deadlines, for instance (database/sql):

```
// ExecContext executes a prepared statement with the given arguments and
// returns a Result summarizing the effect of the statement.
func (s *Stmt) ExecContext(ctx context.Context, args ...interface{}) (Result, error) {
```

- Timeout connections to leave disconnected clients out
- Circuit breakers to protect yourself during incidents

Measure, measure, measure...for debug and alerts

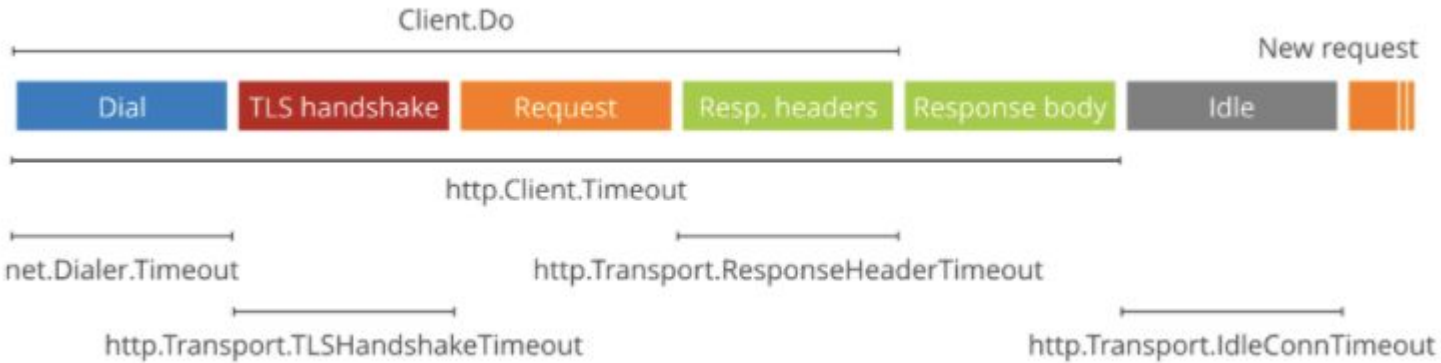
Calling synchronously to other APIs

- Http client timeouts configuration

```
client := &http.Client{Timeout: time.Second * 30}
```

- Retries
- Response error codes - is it a retryable error code? OR there is nothing we can do about it?
  - REST - 429 ?
  - gRPC - INTERNAL ? (https://github.com/grpc/proposal/blob/master/A6-client-retries.md#retryable-status-codes)
  - ...
- Context deadlines - you might be able to subscribe to Done() and cancel?
- HTTP/2 connections multiplexing (i.e. gRPC) - client load balancer? Service mesh?
- Circuit breakers

HTTP Client timeouts

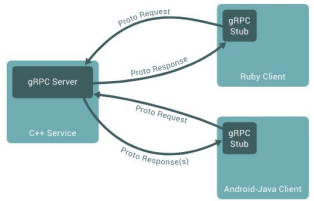


Source: https://blog.cloudflare.com/the-complete-guide-to-golang-net-http-timeouts/

Bootstrap our gRPC API endpoint

- Using Protocol Buffers to serialize messages
- Leverages HTTP/2 advantages
- Besides performance, main advantage is: “You define how you want your data to be structured **once**, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.” (https://developers.google.com/protocol-buffers)
- Mainly used for internal services interaction through sync/async channels (i.e. Apache Kafka)

It's a great capability in our toolbox but comes with some challenges in place such as: debugging (you need to adapt it a bit in order to simply curl to your service :)), client load balancing, etc



gRPC definitions & code generation

```proto
syntax = "proto3";
option go_package = "github.com/eran-levy/tokenizer-gophercon/pkg/proto/tokenizer";
package tokenizer;

message TokenizePayloadRequest {
    string global_tx_id = 1;
    string user_id = 2;
    string organization_id = 3;
    string text = 4;
}

message TokenizePayloadReresponse {
    string global_tx_id = 1;
    repeated string tokenized_text = 2;
    string language = 3;
}
service Tokenizer {
    rpc GetTokens (TokenizePayloadRequest) returns (TokenizePayloadReresponse) {
    }
}
```

```
import (
    _ "google.golang.org/grpc/cmd/protoc-gen-go-grpc"
    _ "google.golang.org/protobuf/cmd/protoc-gen-go"
)
```

```
#!/bin/sh
# run protoc in ../pkg/proto/tokenizer
#possible to use --go-grpc_opt=requireUnimplementedServers=false
protoc --go_out=. --go_opt=paths=source_relative --go-grpc_out=. --go-grpc_opt=paths=source_relative tokenizer.proto
```

gRPC server policies

- Retry policies, health checks, keepalive and others exist here: https://github.com/grpc/grpc-go/blob/master/examples/features
- HTTP/2 connection multiplexing is challenging
- Health updates (use health/v1 api)

```
type serverOptions struct {
    creds                 credentials.TransportCredentials
    codec                 baseCodec
    cp                    Compressor
    dc                    Decompressor
    unaryInt              UnaryServerInterceptor
    streamInt             StreamServerInterceptor
    chainUnaryInts        []UnaryServerInterceptor
    chainStreamInts       []StreamServerInterceptor
    inTapHandle           tap.ServerInHandle
    statsHandler          stats.Handler
    maxConcurrentStreams  uint32
    maxReceiveMessageSize int
    maxSendMessageSize    int
    unknownStreamDesc     *StreamDesc
    keepaliveParams       keepalive.ServerParameters
    keepalivePolicy       keepalive.EnforcementPolicy
    initialWindowSize     int32
    initialConnWindowSize int32
    writeBufferSize       int
    readBufferSize        int
    connectionTimeout     time.Duration
    maxHeaderListSize     *uint32
    headerTableSize       *uint32
    numServerWorkers      uint32
}
```

google.golang.org/grpc@v1.34.0/server.go

Time to write our gRPC API adapter and port to service

gRPC client policies

Retry, load balancing, servers health, context deadlines - https://github.com/grpc/grpc-go/blob/master/examples/features

```
// see https://github.com/grpc/grpc/blob/master/doc/service_config.md to know more about service config
retryPolicy = `{
        "methodConfig": [{
          "name": [{"service": "grpc.examples.echo.Echo"}],
          "waitForReady": true,
          "retryPolicy": {
                  "MaxAttempts": 4,
                  "InitialBackoff": ".01s",
                  "MaxBackoff": ".01s",
                  "BackoffMultiplier": 1.0,
                  "RetryableStatusCodes": [ "UNAVAILABLE" ]
          }
        }]}`
```

We presented here synchronous APIs, but...

- Distributed systems utilize asynchronous APIs heavily, for example: subscribe to Apache Kafka topic
- Events are your API contract!
- Events can be of different types such as: command, domain, etc.
- Concurrently processing traffic is critical but can be complex in the other hand (i.e. what happens if 2 user id events process concurrently?)
- Asynchronous APIs have to react differently to errors, such as: what happens if I couldn't process a given event? Shall I fail completely? Can I retry later and continue processing backpressure? What happens in case of failures that pass a certain threshold? When do I commit that I processed successfully the event?

Our time is limited so we won't be able to further practice it...

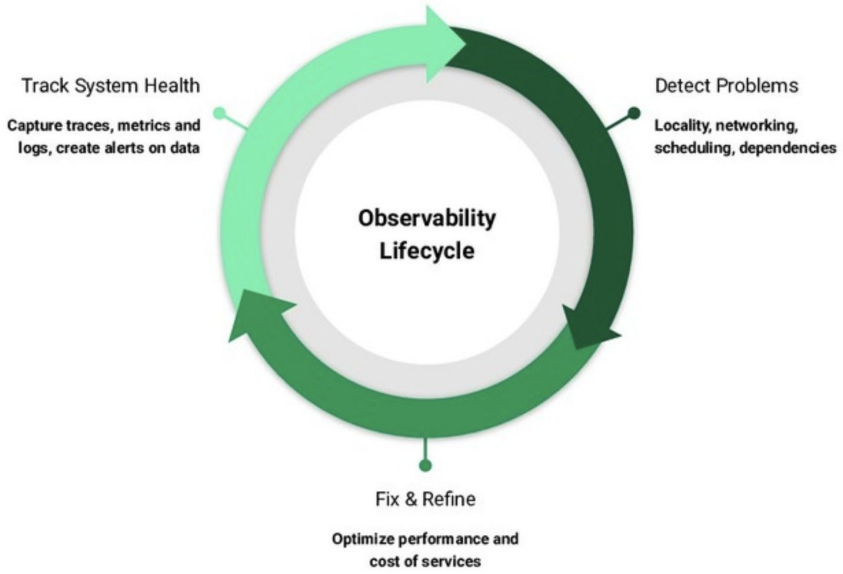
Quick break

Observability

“When environments are as **complex** as they are today, simply **monitoring for known problems** doesn’t address the growing number of new issues that arise. These new issues are “**unknown unknowns**,” meaning that without an observable system, you don’t know what is causing the problem and you don’t have a standard starting point/graph to find out.” (honeycomb.io)

It’s an interesting and huge topic by itself, we will look at it from Go point-of-view

Ultimate Recipe for reliable cloud service



@simon_zeltser

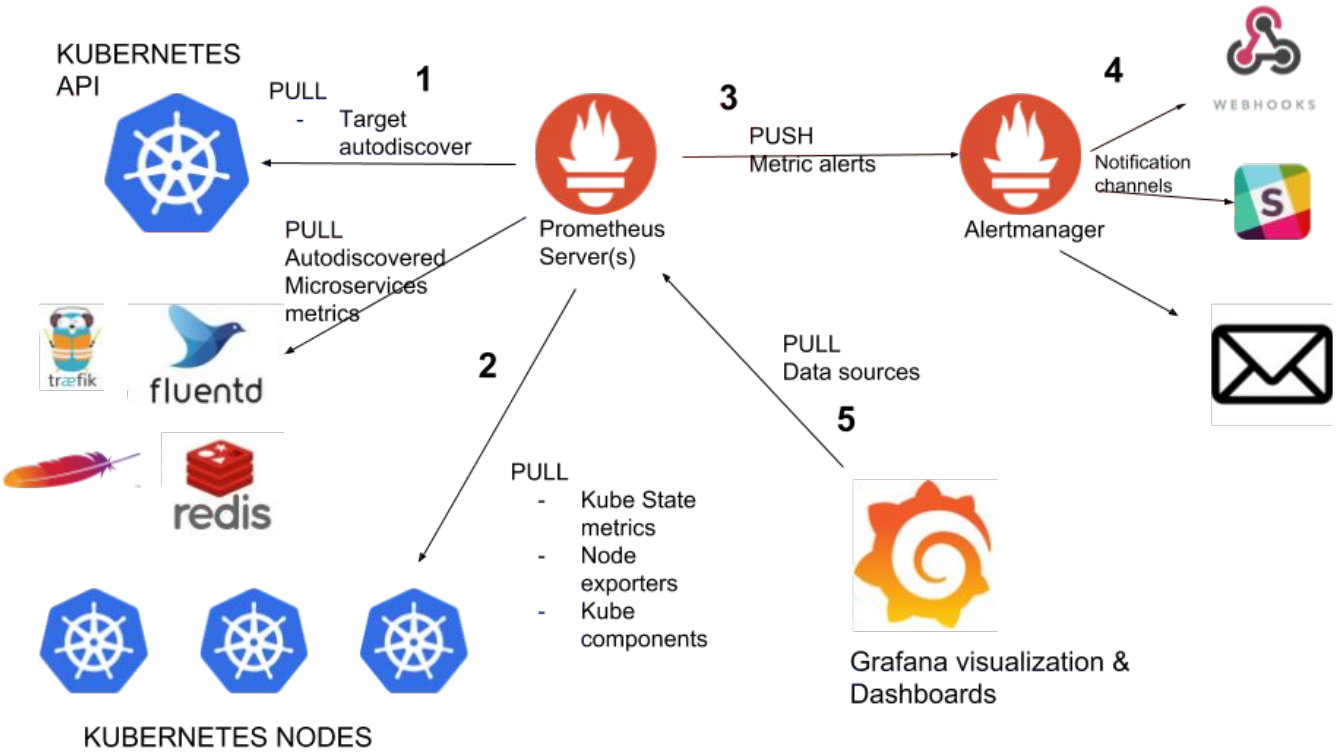
Logging, metrics and traces are your raw data sources

Logging

- Search for a specific pattern in a given time-window or dig into application specific logs
- Write logs to stdout/stderr and the k8s cluster shall take care of the shipping to a central logging infrastructure
- Pick the right package for your need:
  - Built-in “log” package - not structured, not leveled, mostly for dev - std log with timestamp
  - Logrus - JSON format, structured, leveled, hooks (note hooks lock)
  - uber-go/zap - fast (benchmarks: https://github.com/uber-go/zap/tree/master/benchmarks), structured, leveled - performance focused - string formatting, reflection and small allocations are CPU-intensive
  - golang/glog - if performance and volume are highly important, you might consider this one - didn’t get the chance to use

Metrics

- **Metrics** provide quantitative information about processes running inside the system, including counters, gauges, and histograms (OpenTelemetry)
- Measure **business impact** and user experience -
    - Add **custom metrics**
    - build **dashboards**
    - generate **alerts**
- “The four golden signals of monitoring are **latency, traffic, errors, and saturation**.” (Google SRE)
- Metric are used for alerts, circuit breakers, canary release and much more..
- Modern metrics are stored in a time-series database - metric name and key/value tags that create multi-dimensional space
  grpc_io_server_server_latency_count{grpc_server_method="tokenizer.Tokenizer/GetTokens"} 7

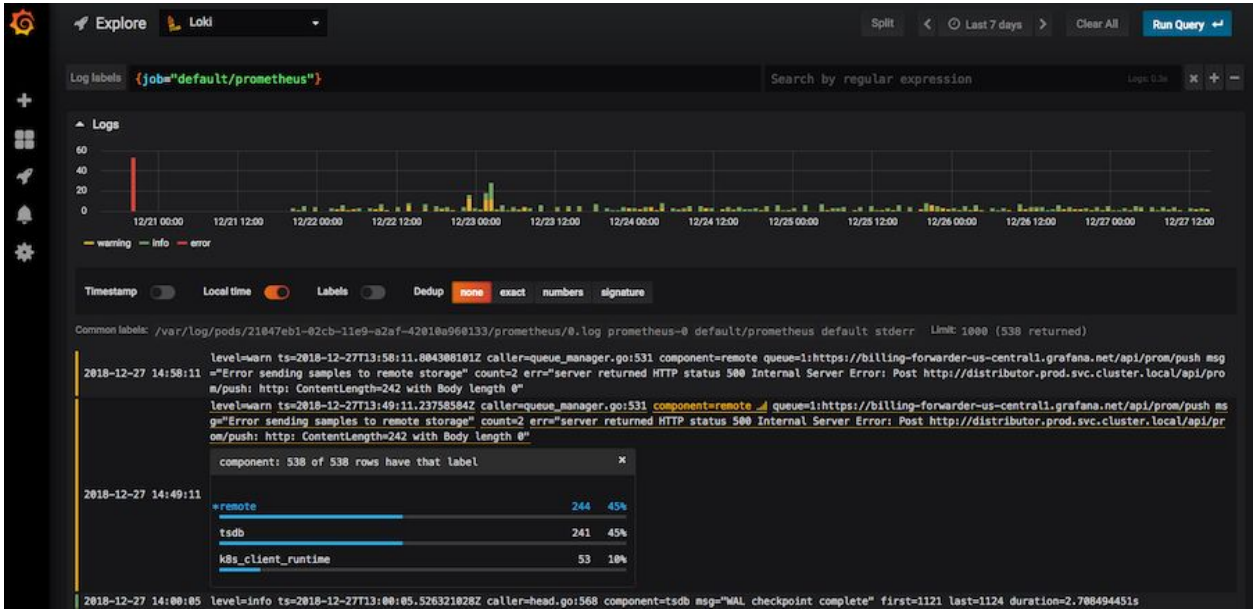


source: sysdig.com

Metrics - Integrate your backend of choice

- Prefer using vendor neutral APIs such as OpenTelemetry to dedicated stats backend clients (i.e. Prometheus go sdk)
- Metrics aren’t sampled - you would like to spot percentile latencies i.e. 99P
- Client libraries usually aggregate the collected metrics data in-process and send to the backend server (prometheus, stackdriver, honeycomb, others)
- Standardize your KPIs to build meaningful dashboards

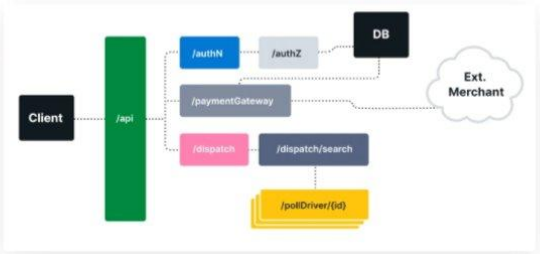
High cardinality labels in metrics challenging



Tracing - Just before we dig in

Figure 1

Figure 2

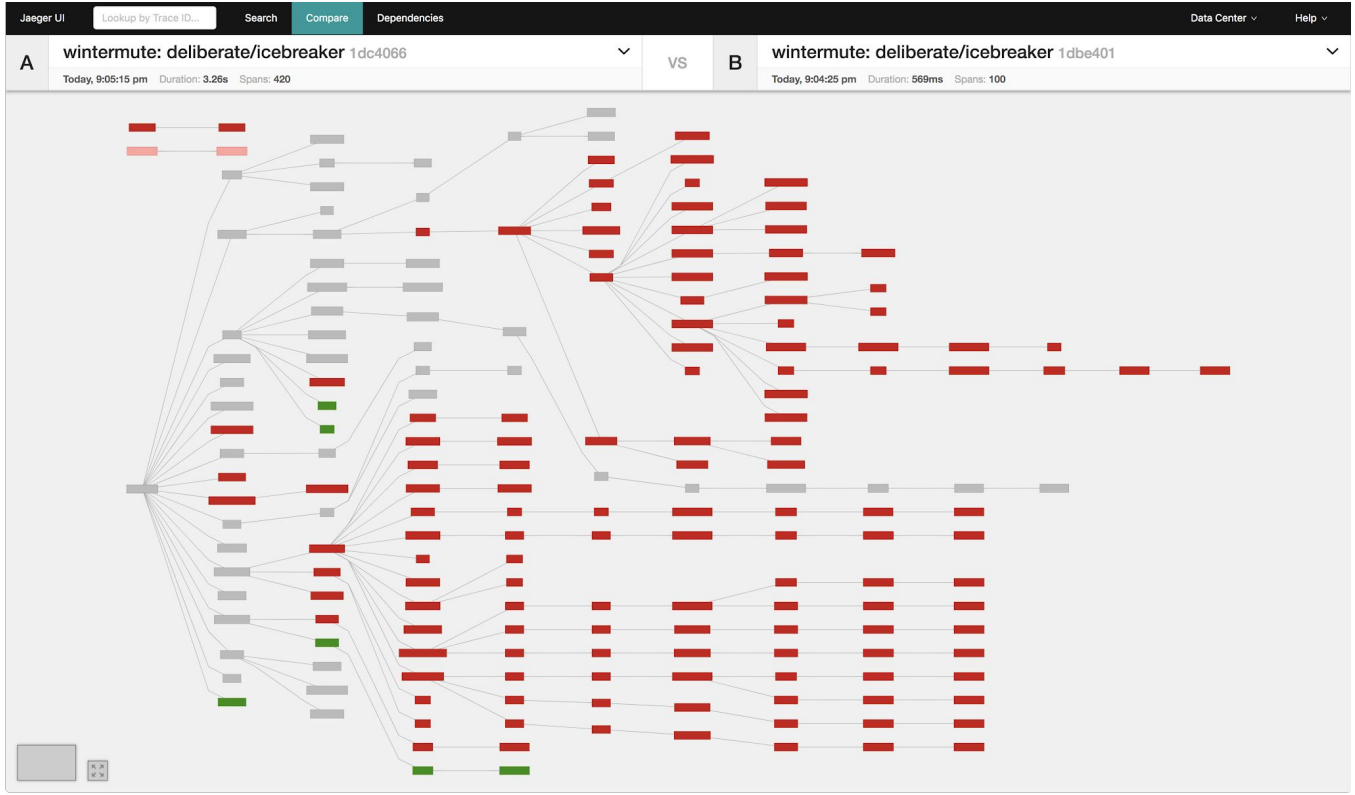


Logical Service Diagram

Service Trace

Source: lightstep

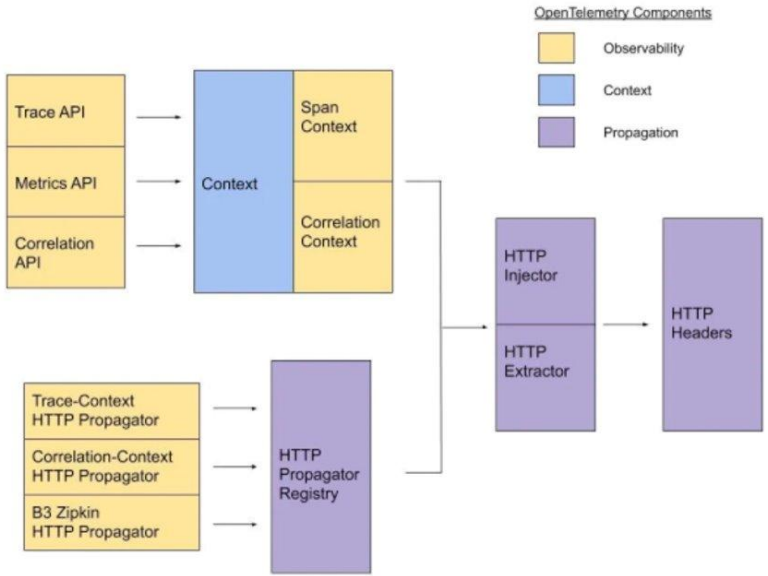
Jaeger


Jaeger UI
Lookup by Trace ID...
Search
Compare
Dependencies
Data Center
Help
A
wintermute: deliberate/icebreaker 1dc4066
Today, 9:05:15 pm Duration: 3.26s Spans: 420
VS
B
wintermute: deliberate/icebreaker 1dbe401
Today, 9:04:25 pm Duration: 569ms Spans: 100

OpenTelemetry

- Opencensus & Opentracing merged into a single project – OpenTelemetry
- OpenTelemetry is a vendor-neutral framework
- As part of the framework, it provides an auto instrumentation for popular libraries (i.e. go-redis, gin)
- OpenTelemetry supports Traces/Metrics (and logs but mostly around Span logs)

Context propagation



Source: lightstep

Quick break

Cache

- Local cache vs distributed cache
- It really depends on your use case, in general try to keep it local as much as possible
- Cache eviction policies - TTL, LRU, others

Map ? hashicorp/golang-lru? dgraph-io/ristretto? patrickmn/go-cache? go-redis?

Persistence

- database/sql standard library and database drivers (i.e. https://github.com/go-sql-driver/mysql, https://github.com/jackc/pgx) - it really depends which database you are using
- You might want to use http://jmoiron.github.io/sqlx/ - sqlx is a package for Go which provides a set of extensions on top of the excellent built-in database/sql package
- http://go-database-sql.org/ - can help you to get started
- Database migration tools (i.e. pressly/goose - Goose is a database migration tool. Manage your database schema by creating incremental SQL changes or Go functions)

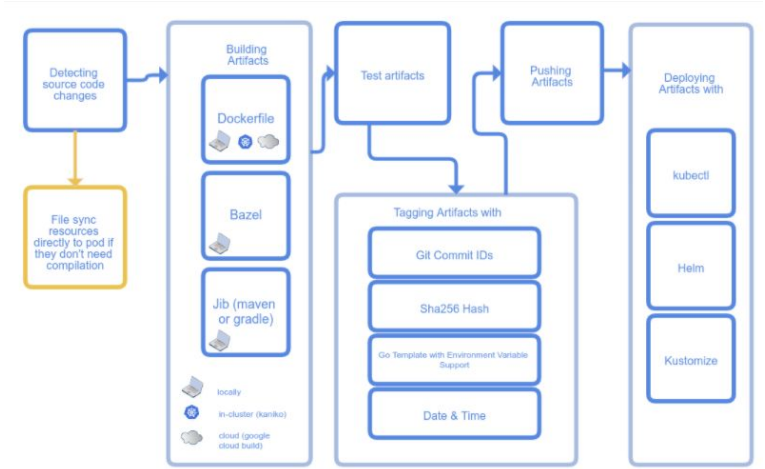
Few things to keep in mind that might important for any client/driver you are integrating in ->

Establishing connections to peripheral services

- Establishing connections is expensive - the clients that you use might utilize connection pools, idle timeouts, retries and other strategies to help you in that
- Network can get slow or experience issues that in high traffic can cause connection failures
- Idle connections and open connections == peak hour connection usage
- Creating connection is expensive
- Rely on context for timeouts
- Cleanup resources! Close connections, shutdown, etc.

Containerization & Deployment

- Containerize our service to release to other environments
- Skaffold can enhance for you the development cycle using a local Kubernetes cluster - feedback loop is fast



Source: skaffold.dev

Summary

- Cloud native engineer design it, develop it, release it, support it
- Cloud native environment are dynamic - our services and development cycles adapt to it
- Keep in mind the fallacies of distributed computing - as engineers we deal with challenging environments these days
- Your synchronous and asynchronous are your contracts with other services (not just customers)
- Observability is a key

Thank you!

Eran Levy
@levyeran