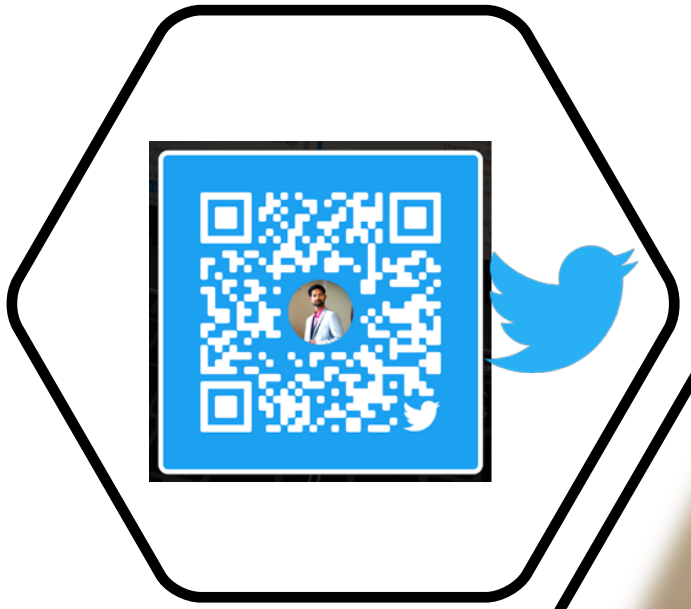
Build Version and gRPC Health Checks

In GoLang Containers



About me

Karthik is a CNCF Speaker, DevOps Architect and Cloud Native App Developer who is passionate about Cloud and Cloud-Native Infrastructure, Developer Tools & Experience and Open-Source Enthusiast.

Karthik is also a passionate writer who writes about the technologies and technology challenges.

Build Versions

Build versions are reference numbers the artifact has been built off from.

Useful when a bug being reported by end-user or if an issue picked up in some functionality; e.g., Integration

It will be easier if the build version can be obtained from the log statements

https://himynameistim.com/blog/the-importance-of-build-numbers

Additional advantages

Being able to reference a specific version that has been pen tested

Referencing a version that's been tested with integrations

Having approval to release a specific version rather than just the latest on master

Anywhere you want to have a conversation referencing releases

How to get version in Go Containers ?

It’s tricky to get the build version into the container images.

Can’t hardcode the build version in the codebase.

Let’s have a main program like this

```
package main

import "github.com/sirupsen/logrus"

var BuildVersion = "development"

func main() {

   logrus.Infof("The version of the app built with 'go build' :
%s\n", BuildVersion)

}
```

Compile, Build, Run

The resulting output from the binary will be

```
INFO[0000] The version of the app built with ‘go build’ : development
```

Linker Flags

Used to pass in flags to underlying linker in Go Toolchain

go build -ldflags="-flag”

https://golang.org/cmd/link/#hdr-Command_Line

```
package main

import "github.com/sirupsen/logrus"

var BuildVersion = "development"

func main() {

   for i:=0; i<2; i++ {
      logrus.Infof("The version of the app built with 'go build' :
%s\n", BuildVersion)
   }

}
```

```
go build -ldflags="-X main.BuildVersion=0.1.0"
```

```
INFO[0000] The version of the app built with 'go build' : 0.1.0
INFO[0000] The version of the app built with 'go build' : 0.1.0
```

@gkarthics

In Docker Image

- Docker have conveniently introduced a feature in 17.05 version called ARG.
- ARG is used to declare a variables as environment variable; accessed across layers of the docker build.
- The value to this variable can be passed during the build time.

```
FROM golang:alpine3.11
ARG BUILD_VERSION
RUN mkdir -p /usr/local/src
COPY . /usr/local/src
WORKDIR /usr/local/src/
RUN go build -ldflags "-X main.BuildVersion=$BUILD_VERSION" .
CMD ./go-build-versioning
```

That’s for the Build versioning in Docker Container Images.

To run this in CI with GitHub, GitLab and Azure ADO scan the QR to read the articles.

HealthChecks in GoLang gRPC Containers for Kubernetes

@gkarthics

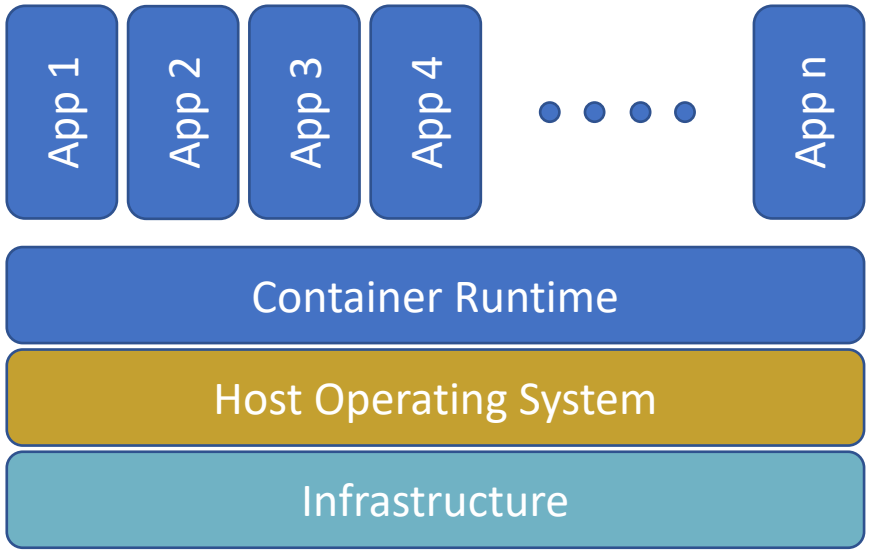
HealthChecks in gRPC Containers

Kubernetes?

- managing containerized workloads

Containers?

- abstractions provides a light-weight OS level virtualization

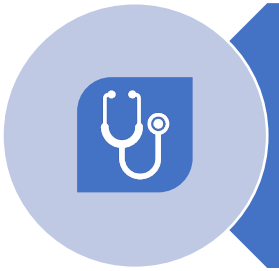


gRPC?

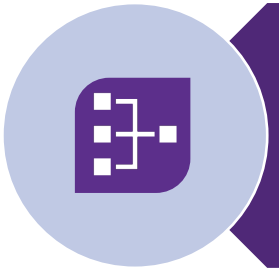
open-source high performance Remote Procedure Call (RPC) framework

@gkarthics

Probes

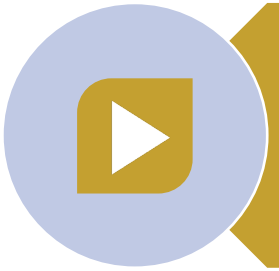


A *Probe* is a diagnostic performed periodically on a Container.



Types of probes

- *startupProbe*
- *readinessProbe*
- *livenessProbe*



Each of these three probes have three different handlers.

- *ExecAction*
- *TCPSocketAction*
- *HttpGetAction*

GRPC Health Checking Protocol

Its clear with gRPC these action handlers will not help.

```
syntax = "proto3";

package grpc.health.v1;

message HealthCheckRequest {
  string service = 1;
}

message HealthCheckResponse {
  enum ServingStatus {
    UNKNOWN = 0;
    SERVING = 1;
    NOT_SERVING = 2;
  }
  ServingStatus status = 1;
}

service Health {
  rpc Check(HealthCheckRequest) returns (HealthCheckResponse);

  rpc Watch(HealthCheckRequest) returns (stream HealthCheckResponse);
}
```

Simple Greet service proto

Let’s look into a simple Greet service proto file

```
1   syntax="proto3";
2
3   option go_package = "proto";
4
5   message HelloRequest {
6       string hello = 1;
7   }
8
9   message HelloResponse {
10      string greet = 1;
11  }
12
13  service GreetService {
14      rpc Hello(HelloRequest) returns(stream HelloResponse) {};
15  }
```

Proto Generation

After running the proto generation command

```
protoc --go_out=plugins=grpc:. proto/hello.proto
```

Let's mimic streaming server by implementing a simple loop for *Hello*

```
1  func (s *server) Hello(helloReq *proto.HelloRequest, srv proto.GreetService_HelloServer) error {
2    logrus.Infof("Server received an rpc request with the following parameter %v", helloReq.Hello)
3
4    for i := 0; i<=10 ; i++ {
5      resp := &proto.HelloResponse{
6        Greet: fmt.Sprintf("Hello %s for %d time",helloReq.Hello, i),
7      }
8
9      srv.SendMsg(resp)
10
11   }
12   return nil
13 }
```

HealthCheck Impl

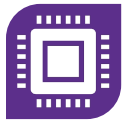
Lets send the status as serving if the end point is available

```
type HealthChecker struct{}

func NewHealthChecker() *HealthChecker {
    return &HealthChecker{}
}

func (s *HealthChecker) Check(ctx context.Context, req *grpc_health_v1.HealthCheckRequest)
    (*grpc_health_v1.HealthCheckResponse, error) {

    logrus.Info("Serving the Check request for health check")

    return &grpc_health_v1.HealthCheckResponse{
        Status: grpc_health_v1.HealthCheckResponse_SERVING,
    }, nil
}
```

Registering HealthCheck service

To register HealthCheck service add the following statement

```
1    healthService := NewHealthChecker()
2    grpc_health_v1.RegisterHealthServer(grpcServer, healthService)
```



The ***grpcServer*** is the new gRPC server that's instantiated and registered for the proto file discussed earlier.

Validate using ***grpc-health-probe*** binary from GRPC Ecosystem

grpc-health-probe -addr "<addr>:<port>"

Dockerfile

We will use the same dev tool for HealthCheck in container as well

To do so, the ***grpc-health-probe*** needs to be bundled with the app

The Dockerfile for it would look like similar to below

```
FROM golang:1.16 as builder
WORKDIR /usr/local/src/
COPY . ./

# Not building the client program
RUN CGO_ENABLED=0 go build -o ./hello-server ./server

# Adding the grpc_health_probe
RUN wget -qO/bin/grpc_health_probe \
    https://github.com/grpc-ecosystem/grpc-health-probe/releases/download/v0.4.0/grpc_health_probe-linux-amd64 && \
    chmod +x /bin/grpc_health_probe


# Final distroless image
FROM gcr.io/distroless/static
COPY --from=builder /usr/local/src/hello-server /hello-server
COPY --from=builder /bin/grpc_health_probe ./grpc_health_probe
CMD ["/hello-server"]

```

Kubernetes Probes

We have the image built, lets define the probes for it.

Since we have included the cli tool in the image, the “ExecAction” can be invoked to make use of it.

The syntax of the probe definition in the Kubernetes yaml spec will look like

```yaml
spec:
  containers:
    - image: gkarthics/grpc-helth-check-tutorial:0.1.0
      imagePullPolicy: Always
      name: grpc-health-check
      ports:
        - containerPort: 5000
      readinessProbe:
        exec:
          command: ["/grpc_health_probe", "-addr=:5000"]
        initialDelaySeconds: 5
      livenessProbe:
        exec:
          command: ["/grpc_health_probe", "-addr=:5000"]
        initialDelaySeconds: 10
```

For a detailed step by step guide, scan the QR

@gkarthics

Thank You!!!

Know more about me!

@gkarthics