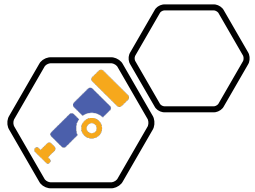
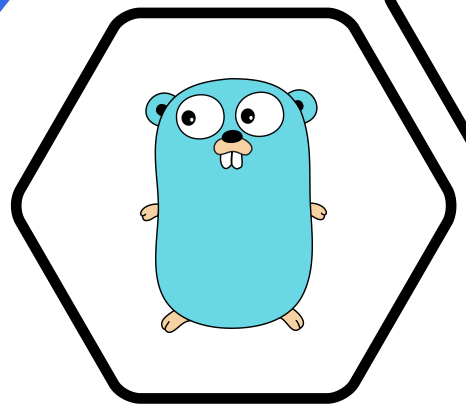
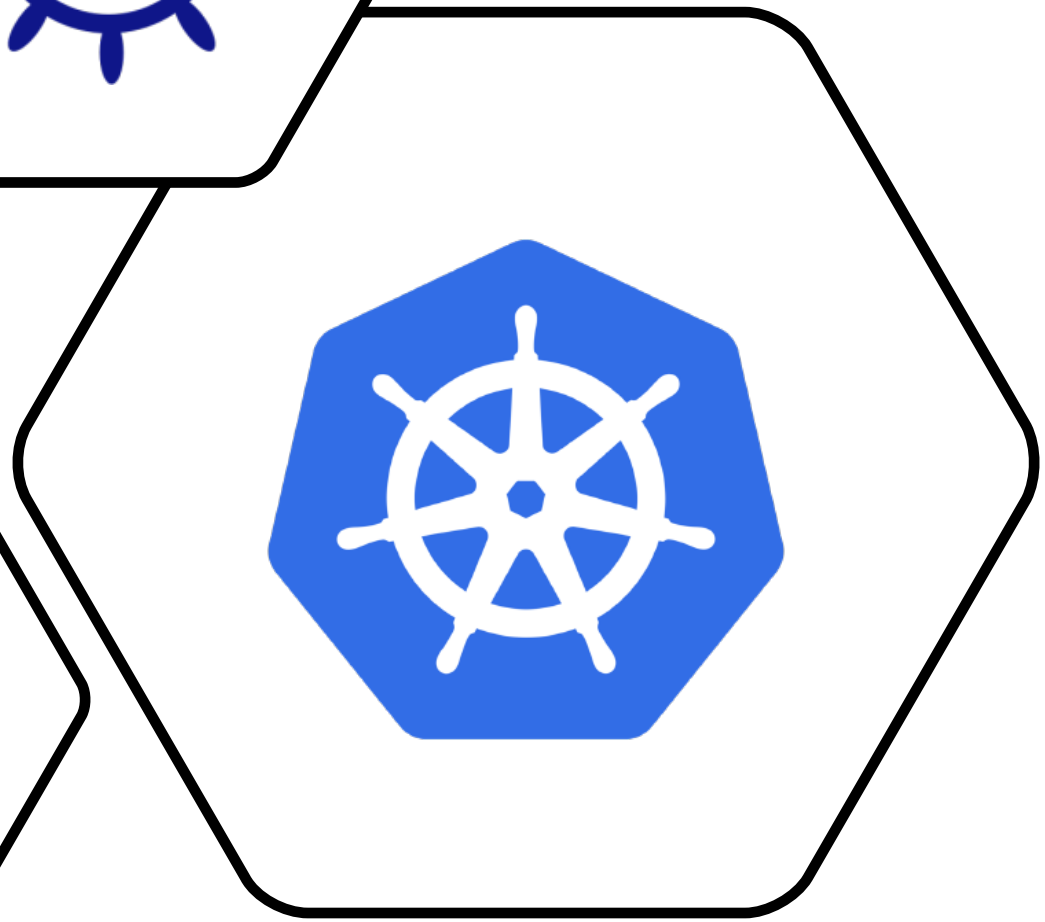


Develop(), Deploy() and Monitor Cloud-Native App in Kubernetes





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.

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

Karthikeyan Govindaraj

Know more about me!!!

@gkarthics

What is Cloud-Native

- Cloud-native is an approach to building and running applications that exploit the advantages of the cloud computing delivery model.

- ***CNCF*** *defines cloud-native as "scalable applications" running in "modern dynamic environments" that use technologies such as containers, microservices, and declarative APIs.*

GoLang

- open-source programming language by Google
- highly performant
- highly expressive
- good readability
- both statically and dynamically typed
- features of C++ , Java and Python
- a large part of tools for cloud (Docker, Kubernetes) is also written in Go

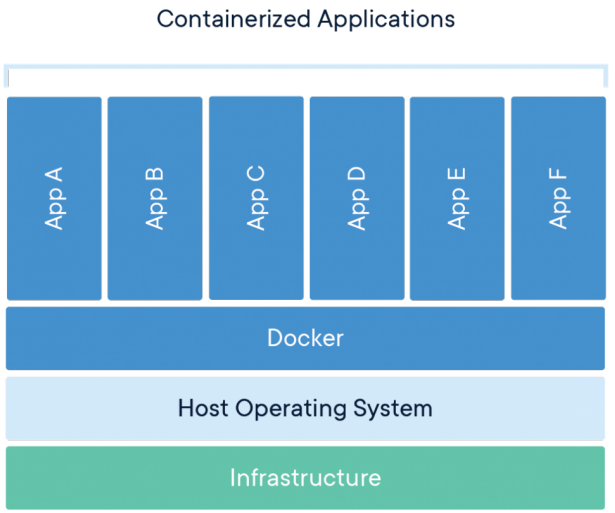
OpenTelemetry

OpenTelemetry is a collection of tools, APIs, and SDKs. Use it to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) to help you analyze your software's performance and behavior.

Container

A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another.

- Agility
- Portability
- Rapid scalability



Kubernetes

Kubernetes is a portable, extensible, open-source platform.

Manages containerized workloads and services.

Facilitates both declarative configuration and automation

Service discovery and load balancing

Storage orchestration

Automated rollouts and rollbacks

Self-healing

Secret and configuration management

minikube is local Kubernetes, focusing on making it easy to learn and develop for Kubernetes.

@gkarthics

Helm

The package manager for Kubernetes

Helm Charts help you define, install, and upgrade even the most complex Kubernetes application.

Charts are easy to create, version, share, and publish

```go
func main() {
    http.HandleFunc("/ping", ping)

    http.ListenAndServe(":8090", nil)
}

```

```go
func ping(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, format: "pong\n")
}
```

Let’s start the Simple Webserver

@gkarthics

Add otel modules

go get	go.opentelemetry.io/otel/sdk/trace
go get	go.opentelemetry.io/otel/exporters/stdout/stdouttrace
go get	go.opentelemetry.io/otel/propagation
go get	go.opentelemetry.io/otel/semconv

https://github.com/open-telemetry/opentelemetry-go/tree/main/exporters/

Create a tracer

```go
func initTracer() *sdktrace.TracerProvider {
    exporter, err := stdout.New(stdout.WithPrettyPrint())
    if err != nil {
        log.Fatal(err)
    }

    tp := sdktrace.NewTracerProvider(
        sdktrace.WithSampler(sdktrace.AlwaysSample()),
        sdktrace.WithBatcher(exporter),
        sdktrace.WithResource(resource.NewWithAttributes(semconv.SchemaURL,
            semconv.ServiceNameKey.String("Service-1"))),
    )
    otel.SetTracerProvider(tp)
    otel.SetTextMapPropagator(propagation.NewCompositeTextMapPropagator(propagation.TraceContext{},
        propagation.Baggage{}))
    return tp
}
```

Add tracer to main function

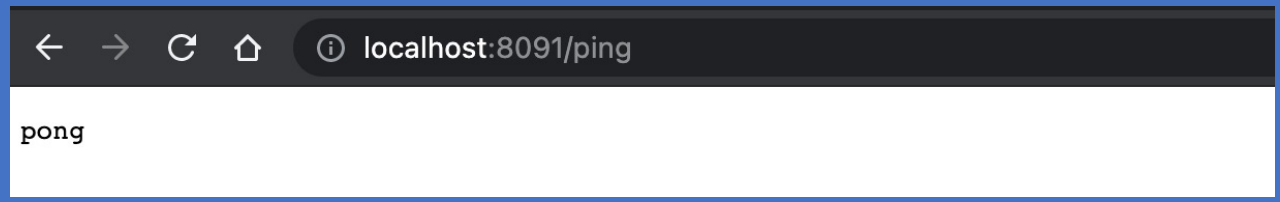
```
1  func main() {
2      tp := initTracer()
3
4      defer func() {
5          if err := tp.Shutdown(context.Background()); err != nil {
6              log.Printf("Error shutting down tracer provider: %v", err)
7          }
8      }()
9
10     http.Handle("/ping", NewHandler(http.HandlerFunc(ping), "ping"))
11
12     http.ListenAndServe(":8090", nil)
13 }
```

Handle the HTTP Func via OTEL

```
http.Handle("/ping",
    otelhttp.NewHandler(http.HandlerFunc(ping),
        "ping"))
```

...response


```
{
        "Name": "ping",
        "SpanContext": {
                "TraceID": "58e4161c8321b02c3c8078df2988de59",
                "SpanID": "9e56a8fb192fc5db",
                "TraceFlags": "01",
                "TraceState": "",
                "Remote": false
        },
        "Parent": {
                "TraceID": "00000000000000000000000000000000",
                "SpanID": "0000000000000000",
                "TraceFlags": "00",
                "TraceState": "",
                "Remote": false
        },
        "SpanKind": 2,
        "StartTime": "2021-10-09T13:30:43.327335+05:30",
        "EndTime": "2021-10-09T13:30:43.327653743+05:30",
        "Attributes": [
                {
                        "Key": "net.transport",
                        "Value": {
                                "Type": "STRING",
                                "Value": "ip_tcp"
                        }
                },
                {
                        "Key": "net.peer.ip",
                        "Value": {
                                "Type": "STRING",
                                "Value": "127.0.0.1"
                        }
```

Let’s talk to Kubernetes

- client-go: **Go client** for talking to a kubernetes cluster
  - in-cluster
    - config, err := rest.InClusterConfig()
  - out-of-cluster

```
var kubeconfig *string
if home := homedir.HomeDir(); home != "" {
    kubeconfig = flag.String("kubeconfig", filepath.Join(home, ".kube", "config"),
    "(optional) absolute path to the kubeconfig file")
} else {
    kubeconfig = flag.String("kubeconfig", "", "absolute path to the kubeconfig file")
}
flag.Parse()

config, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)
```

k8s-discovery

- go get github.com/gkarthiks/k8s-discovery
  - Avoids the boiler plate code for the client-go
  - Auto configures the in-cluster or out-of-cluster configuration
  - k8s, _ := discovery.NewK8s()
    - Provides a client attached to the targeted Kubernetes cluster from local or running host cluster if from within a pod

```
type K8s struct {
  Clientset       coreClient.Interface
  MetricsClientSet *metricsClient.Clientset
  RestConfig      *restClient.Config
}
```

List all pods in a cluster

```
func listPods() {
    k8s, _ := discovery.NewK8s()
    var pods []string

    podList, _ := k8s.Clientset.CoreV1().Pods("").List(context.Background(),
                        v1.ListOptions{})

    for _, pod := range podList.Items {
        pods = append(pods, pod.Name)
    }

    fmt.Printf("Pods in the cluster: %v", pods)
}
```

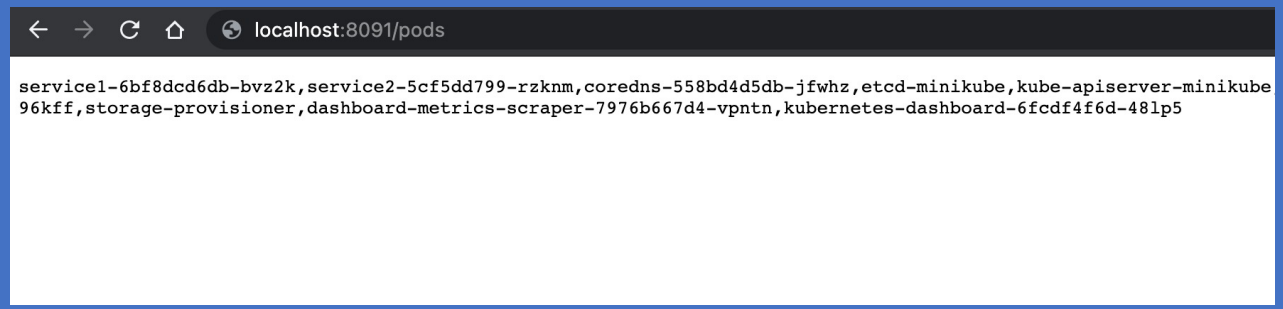
```
http.Handle("/pods",
    otelhttp.NewHandler(http.HandlerFunc(listPods),
        "List Pods"))
```

```
func listPods(w http.ResponseWriter, req *http.Request) {
```

Add this to the HTTP Handler via OTEL

...response


```
{
        "Name": "List Namespaces",
        "SpanContext": {
                "TraceID": "23dfb47ad1dc1dea010ff4f4a65169c7",
                "SpanID": "b04ea0e2cd22f489",
                "TraceFlags": "01",
                "TraceState": "",
                "Remote": false
        },
        "Parent": {
                "TraceID": "00000000000000000000000000000000",
                "SpanID": "0000000000000000",
                "TraceFlags": "00",
                "TraceState": "",
                "Remote": false
        },
        "SpanKind": 2,
        "StartTime": "2021-10-09T14:09:29.806179+05:30",
        "EndTime": "2021-10-09T14:09:29.841334565+05:30",
        "Attributes": [
                {
                        "Key": "net.transport",
                        "Value": {
                                "Type": "STRING",
                                "Value": "ip_tcp"
                        }
                },
                {
                        "Key": "net.peer.ip",
                        "Value": {
                                "Type": "STRING",
                                "Value": "127.0.0.1"
```

```go
package main

import (...
)

func ping(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "pong\n")
}

func initTracer() *sdktrace.TracerProvider {...
}

func main() {...
}

func listNamespace(w http.ResponseWriter, req *http.Request) {...
```

Containerize

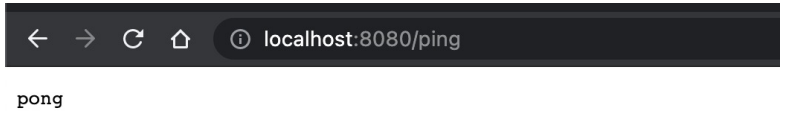
```
FROM golang:1.17-alpine AS 🔨 builder
WORKDIR /go/src/app
COPY . /go/src/app
RUN go build -o service2 main.go


FROM alpine
COPY --from=builder /go/src/app/service2 /bin
CMD ["/bin/service2"]
```

```
docker build . -t service2
```

```
REPOSITORY          TAG       IMAGE ID       CREATED              SIZE
service2            latest    1a45f53272f1   26 seconds ago       51MB
```

...response

```
Sat Oct 09@14:27:22  ~/GSKDeveloper/cwitc-service2 * > docker run -p 8080:8091 service2:latest
{
        "Name": "ping",
        "SpanContext": {
                "TraceID": "b1b383e3cd8cea83ffe4f42a50b0cc15",
                "SpanID": "feecea12ffcc7e3a",
                "TraceFlags": "01",
                "TraceState": "",
                "Remote": false
        },
        "Parent": {
                "TraceID": "00000000000000000000000000000000",
                "SpanID": "0000000000000000",
                "TraceFlags": "00",
                "TraceState": "",
                "Remote": false
        },
        "SpanKind": 2,
        "StartTime": "2021-10-09T08:58:21.3158784Z",
        "EndTime": "2021-10-09T08:58:21.3159499Z",
        "Attributes": [
                {
                        "Key": "net.transport",
                        "Value": {
                                "Type": "STRING",
                                "Value": "ip_tcp"
                        }
                },
                {
                        "Key": "net.peer.ip",
                        "Value": {
                                "Type": "STRING",
                                "Value": "172.17.0.1"
```



Let’s create a parent service

```go
func listNamespace(w http.ResponseWriter, req *http.Request) {
    url := "http://localhost:8091/namespaces"

    ctx := context.Background()
    resp, _ := otelhttp.Get(ctx, url)

    var data []string
    dec := gob.NewDecoder(resp.Body)
    dec.Decode(&data)

    fmt.Println("response Status:", resp.Status)
    fmt.Println("response Headers:", resp.Header)
    fmt.Fprintf(w, strings.Join(data, ","))
}
```

Running a GET namespace from s1 to s2


```
{
    "Name": "HTTP GET",
    "SpanContext": {
        "TraceID": "eb53440f856e3856adc5c3eec4dbdc10",
        "SpanID": "d41c786c6a0b7569",
        "TraceFlags": "01",
        "TraceState": "",
        "Remote": false
    },
    "Parent": {
        "TraceID": "00000000000000000000000000000000",
        "SpanID": "0000000000000000",
        "TraceFlags": "00",
        "TraceState": "",
        "Remote": false
    },

    "Resource": [
        {
            "Key": "service.name",
            "Value": {
                "Type": "STRING",
                "Value": "Service-1"
            }
        }
    ]
}
```

```
{
    "Name": "List Namespaces",
    "SpanContext": {
        "TraceID": "eb53440f856e3856adc5c3eec4dbdc10",
        "SpanID": "9405475772d8a294",
        "TraceFlags": "01",
        "TraceState": "",
        "Remote": false
    },
    "Parent": {
        "TraceID": "eb53440f856e3856adc5c3eec4dbdc10",
        "SpanID": "d41c786c6a0b7569",
        "TraceFlags": "01",
        "TraceState": "",
        "Remote": true
    },
    "Resource": [
        {
            "Key": "service.name",
            "Value": {
                "Type": "STRING",
                "Value": "Service-2"
            }
        }
    ]
}
```

Let’s prepare to deploy

minikube

- Start minikube

eval `minikube docker-env`

- Build images inside the minikube machine.

helm create service1

- specify image as service1:latest in values.yaml
- containerPort as 8090 in deployment.yaml
- Readiness and liveness path as /ping

helm create service2

- specify image as service2:latest in values.yaml
- containerPort as 8091 in deployment.yaml
- Readiness and liveness path as /ping

Demo!!!