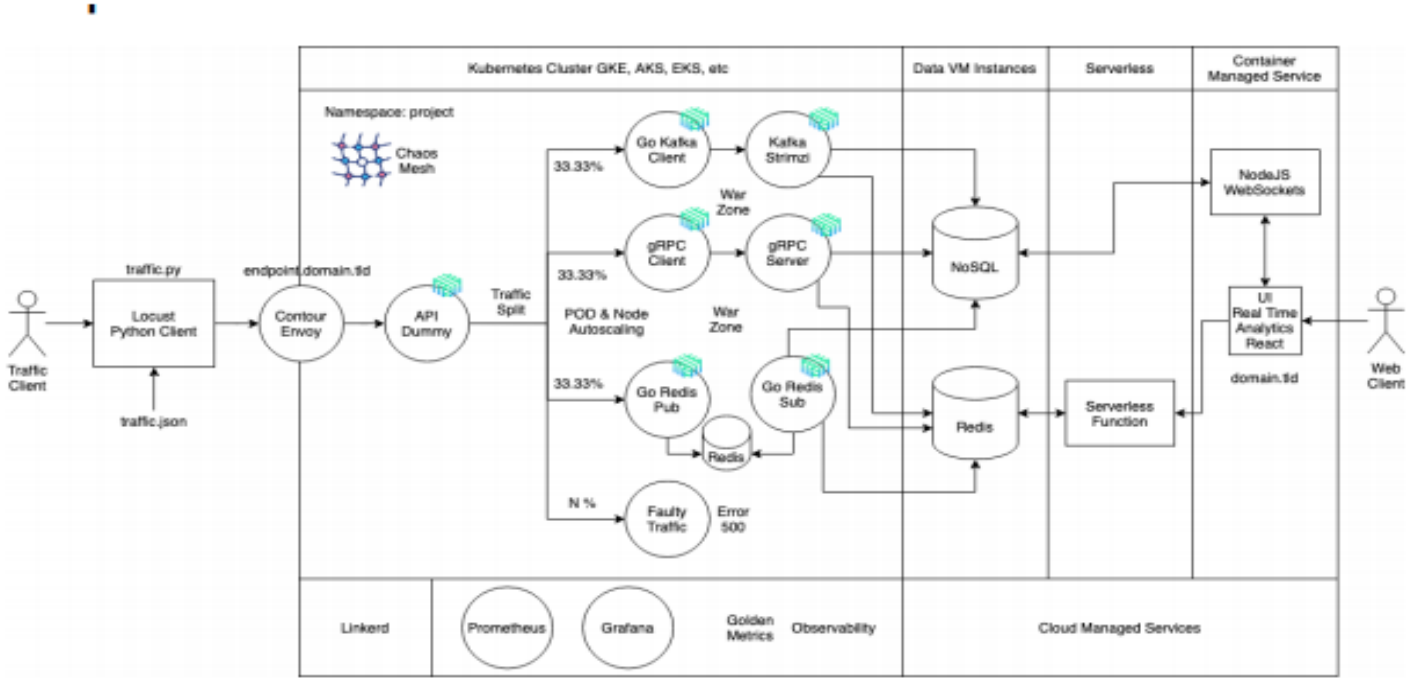
Tecnologias utilizadas:

- DOCKER
- NGINX
- KUBERNETES
- REDIS
- GRPC
- MONGO
- LOCUST
- REACT
- NODE JS

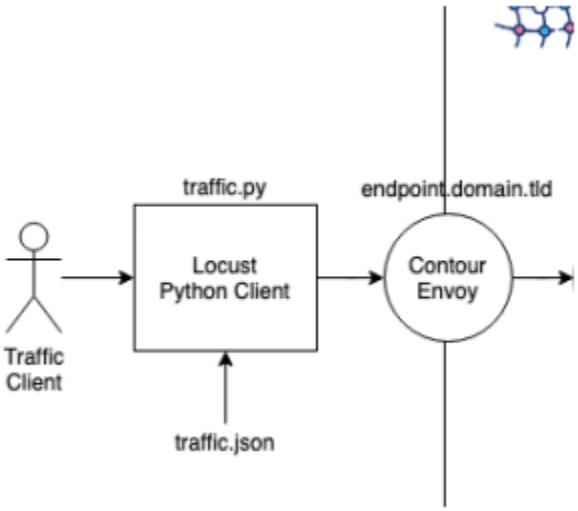
Arquitectura del sistema:

El sistema posee un sistema distribuido que permite que los datos sean ingresados desde distintas fuentes, con la posibilidad de dirigirse por dos distintas rutas para ser almacenados. El primer flujo es llamado “Blue Deployment” utiliza gRPC para comunicarse, y almacenar la información en la base de datos de Mongo. El segundo flujo es redirigido a una cola de mensajes implementada en Redis, que luego almacena permanentemente en Mongo. La información que ingrese al sistema se dividirá en un 50% para cada uno de los flujos. Para consultar la información, existe una interfaz de usuario implementada en React, la cual muestra los datos en tiempo real. Cada una de estas partes debe ser observable y monitoreada con



Primera Parte:

Con el propósito de realizar pruebas de tráfico al sistema, se utilizó el framework de Locust, el cual lee un archivo de entrada, selecciona una persona de manera aleatoria y la envia al sistema, por medio de una solicitud de tipo POST.



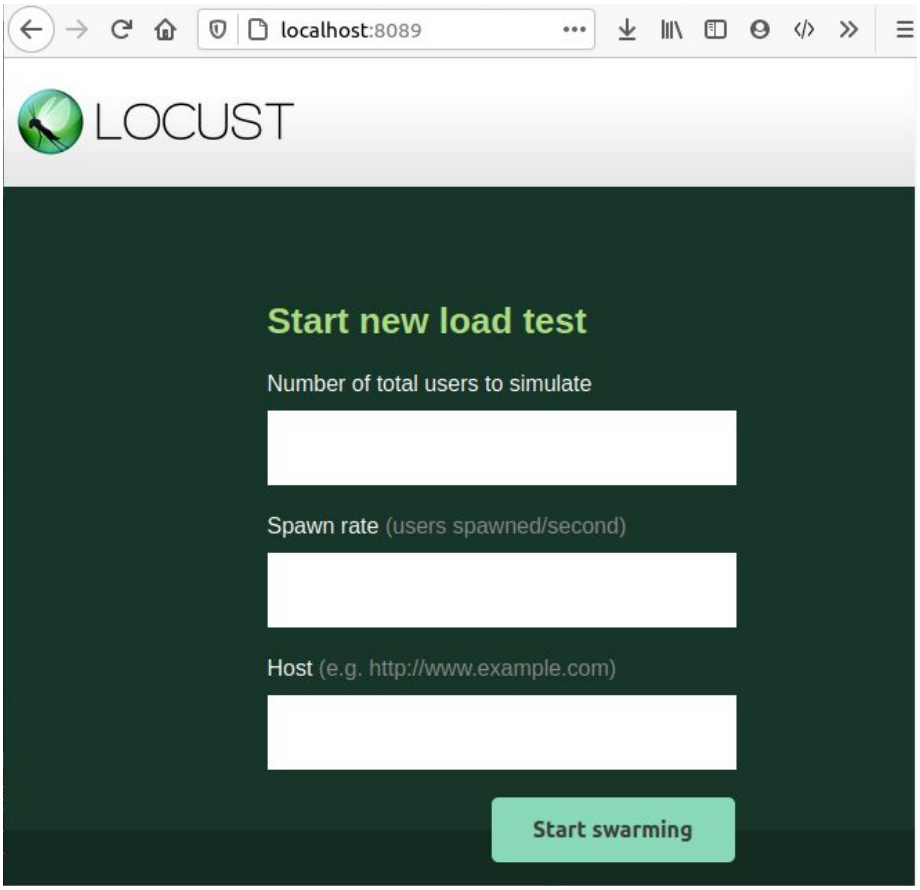
El script es el siguiente:

```
import time, json, random
from locust import HttpUser, task

class TrafficClass(HttpUser):

   @task
   def sendtraffic(self):
       with open('traffic.json', 'r') as f:
           pacientes = json.load(f)
       length = len(pacientes)
       print(length)
       index = random.randint(0,length - 1)
       print(pacientes[index])
       self.client.post("/", json=pacientes[index])
```

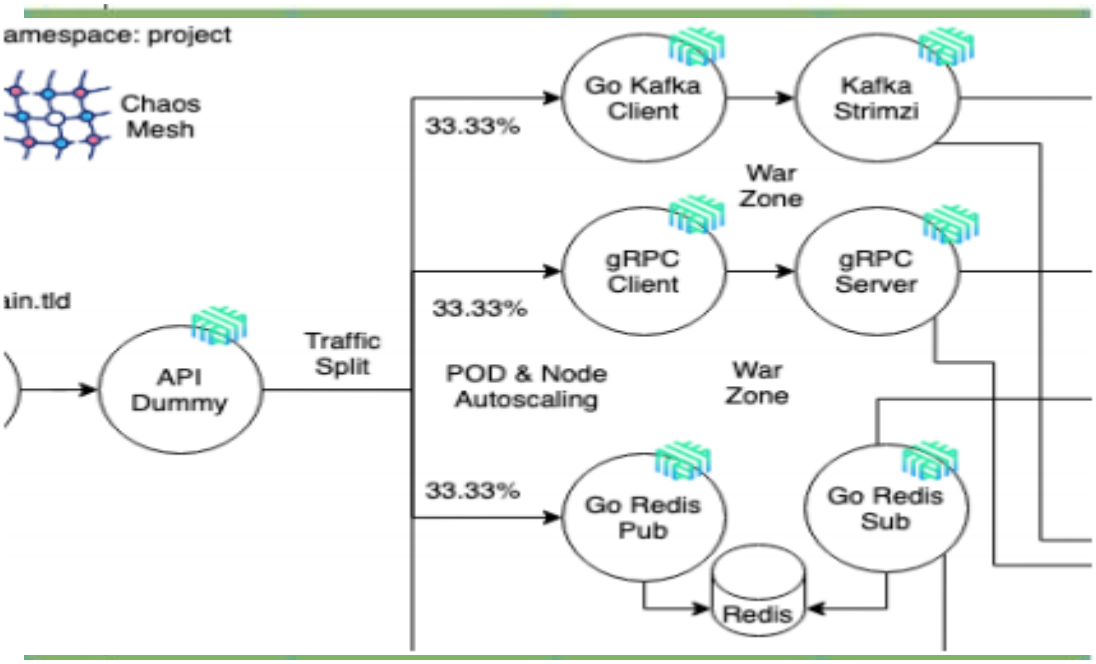
Cabe recalcar que el archivo trafic.json debe estar en el mismo directorio, para ejecutar el script se debe colocar:

```
    $locust -f traffic.py
```

Esto desplegará un servidor que estará escuchando en el puerto 8089 y se verá de la siguiente manera:



Segunda Parte:



GitHub

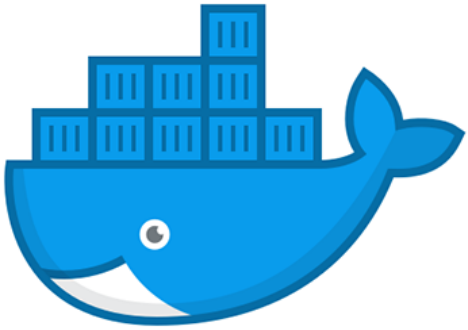
se utiliza como plataforma para almacenar, crear código fuente de proyectos con un control de versiones git

Kubernetes

kubernetes es una plataforma portable de código abierto para administrar cargas de trabajo y servicios, también facilita a la automatización y configuración declarativa.

Docker

Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos.utiliza características de aislamiento de recursos del kernel Linux, tales como cgroups y espacios de nombres para permitir que contenedores independientes se ejecuten dentro de una sola instancia de Linux, evitando la sobrecarga de iniciar y mantener máquinas virtuales.



Tambien se creo un archivo Dockerfile para la creacion de imagenes con docker

```
# Start by building the application.
FROM golang:1.13-buster as build

WORKDIR /go/src/app
ADD . /go/src/app

RUN go get -d -v ./...

RUN go build -o /go/bin/app

# Now copy it into our base image.
FROM gcr.io/distroless/base-debian10
COPY --from=build /go/bin/app /
CMD ["/app"]
```

Las librerias utilizadas en la parte de PubSub son:

```
import (
    "context"
    "encoding/json"
    "fmt"
    "log"

    "github.com/garyburd/redigo/redis"
    "go.mongodb.org/mongo-driver/mongo"
    "go.mongodb.org/mongo-driver/mongo/options"
)
```

Para la coneccion con Pub redis se implementó de la siguiente manera:

```
func main() {
    c, err := redis.Dial("tcp", "18.216.215.34:6379")
    if err != nil {
        fmt.Println("error de conexión a la base de datos redis", err)

    }
    psc := redis.PubSubConn{Conn: c}
    psc.Subscribe("pubsub2")
    for {

        switch v := psc.Receive().(type) {
        case redis.Message:
            var inf Infectado
            json.Unmarshal([]byte(v.Data), &inf)
            mongoInsert(inf.Name, inf.Location, inf.Age, inf.Infected_type, inf.State)
        case redis.Subscription:
            fmt.Printf("%s: %s %d\n", v.Channel, v.Kind, v.Count)
        case error:
            fmt.Printf("error", v)
        }
    }
}
```

para el envio de datos a MongoBD se realizo de la siguiente manera:

```go
func mongoInsert(nombre string, departamento string, edad int, tipo string, estado strin
    clientOptions := options.Client().ApplyURI(uri)
    client, err := mongo.Connect(context.TODO(), clientOptions)

    if err != nil {
        log.Fatal(err)
    }

    err = client.Ping(context.TODO(), nil)

    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("Connected to MongoDB!")

    collection := client.Database("covid").Collection("covids")

    infectado := Infectado{nombre, departamento, edad, tipo, estado}

    //json.Unmarshal([]byte(infectedJson), &infected)
    fmt.Println(infectado)
    insertResult, err := collection.InsertOne(context.TODO(), infectado)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("Inserted a Single Document: ", insertResult.InsertedID)
}
```

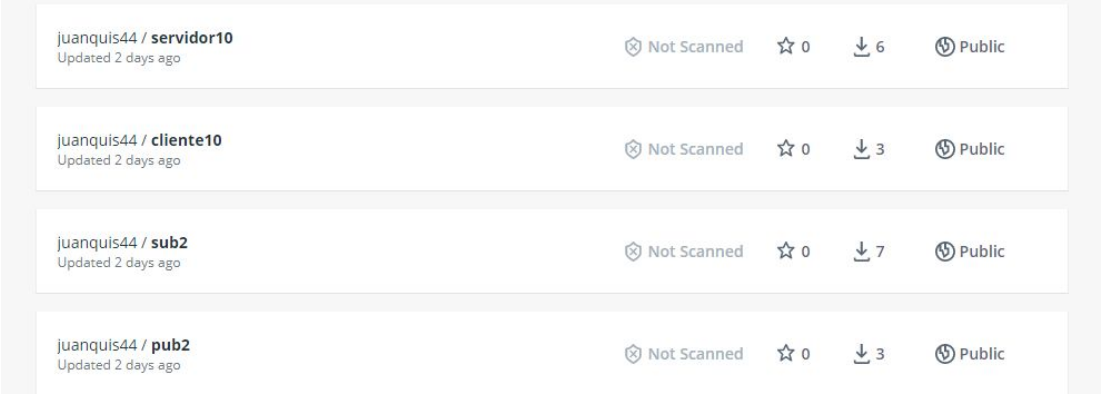
en la parte de Sub redis la coneccion para recibir datos es:

```go
func locus(w http.ResponseWriter, r *http.Request) {
    enableCors(&w) // habilitamos cors
    data, err := ioutil.ReadAll(r.Body)
    if err != nil {
        log.Fatalln(err)
    }
    if data != nil && string(data) != "" {
        aux := string(data)
        aux = strings.Replace(aux, "ñ", "n", len(aux))
        aux = strings.Replace(aux, "Ñ", "N", len(aux))

        respuesta := redisInsert(aux)

        if respuesta != nil {
            fmt.Fprintf(w, "%v", respuesta)
        }
        if respuesta == nil {
            fmt.Fprintf(w, "funciona")
        }
        return
    } else {
        fmt.Fprintf(w, "vacio o nulo")
    }
}
```

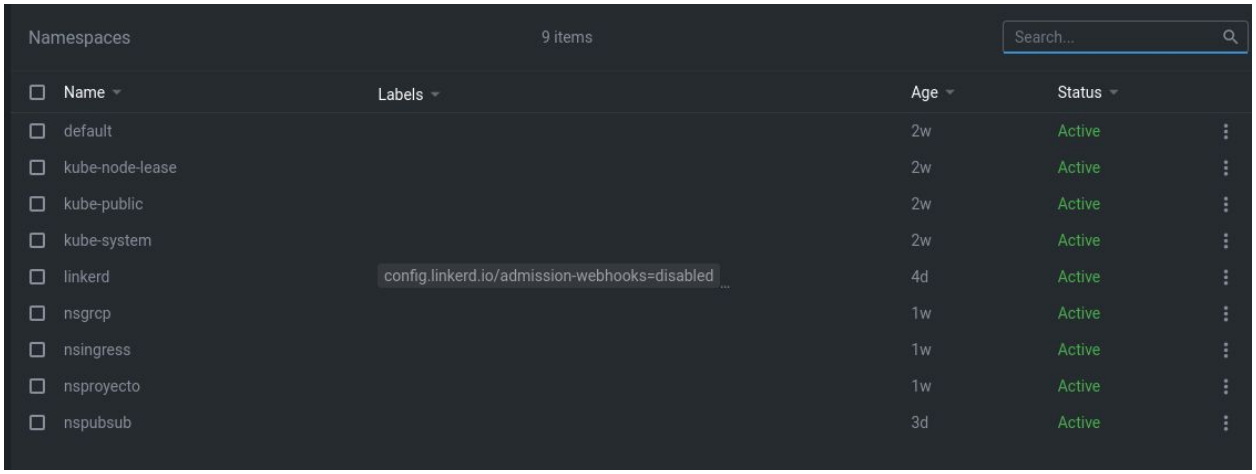
para el envio de datos hacia redisDB:

```go
func redisInsert(datos string) (respuesta error) {

    conn, err := redis.Dial("tcp", "18.216.215.34:6379")
    if err != nil {
        fmt.Println("error de conexión a la base de datos redis", err)
        respuesta = err
        return
    }
    //conn.Send("LPush", "jsondata3", datos)
    //conn.Send("set", "a", "1")
    if string(datos) != "" {
        rep, err := conn.Do("LPush", "jsondata3", datos)
        if err != nil {
            respuesta = err
            return
        }
        fmt.Println("funciona", rep)
        rep1, err1 := conn.Do("PUBLISH", "pubsub2", datos)
        if err1 != nil {
            respuesta = err1
            return
        }
        fmt.Println("funciona", rep1)
    }

    return
```

en la parte de docker Hub las imágenes utilizadas para el proyecto fueron:

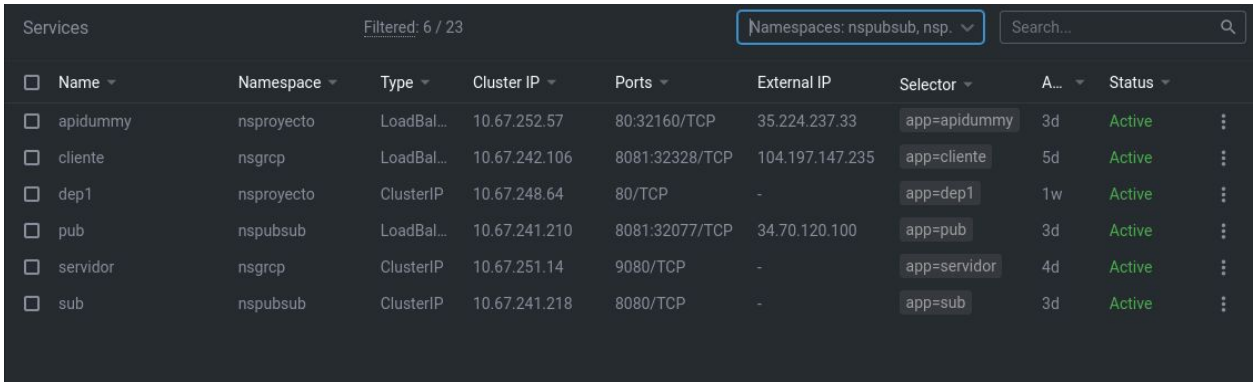


en la parte de kubernetes los servicios namespaces y pods utilizados son los siguientes:



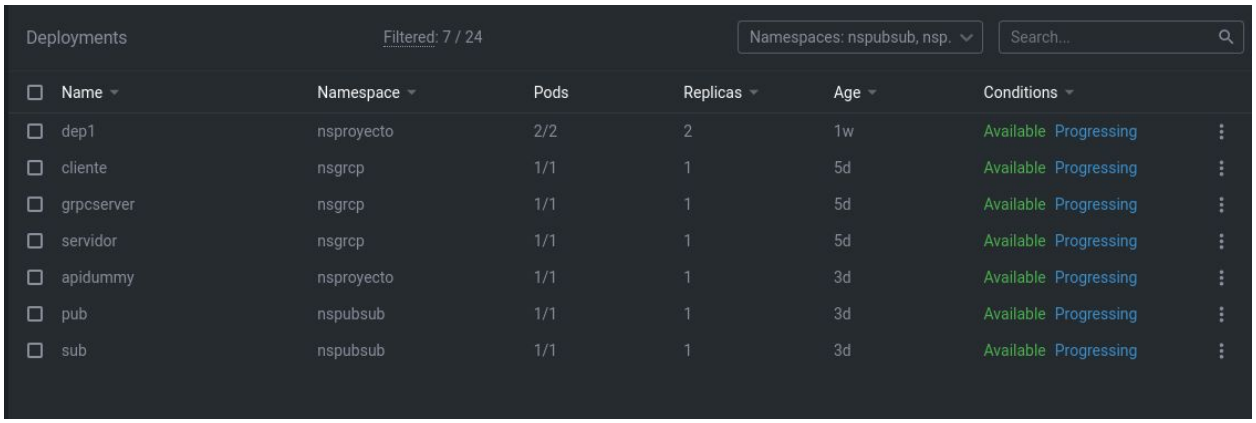
Namespaces — 9 items — Search...

Name	Labels	Age	Status
default		2w	Active
kube-node-lease		2w	Active
kube-public		2w	Active
kube-system		2w	Active
linkerd	config.linkerd.io/admission-webhooks=disabled	4d	Active
nsgrcp		1w	Active
nsingress		1w	Active
nsproyecto		1w	Active
nspubsub		3d	Active



Services — Filtered: 6 / 23 — Namespaces: nspubsub, nsp. — Search...

Name	Namespace	Type	Cluster IP	Ports	External IP	Selector	A...	Status
apidummy	nsproyecto	LoadBal...	10.67.252.57	80:32160/TCP	35.224.237.33	app=apidummy	3d	Active
cliente	nsgrcp	LoadBal...	10.67.242.106	8081:32328/TCP	104.197.147.235	app=cliente	5d	Active
dep1	nsproyecto	ClusterIP	10.67.248.64	80/TCP	-	app=dep1	1w	Active
pub	nspubsub	LoadBal...	10.67.241.210	8081:32077/TCP	34.70.120.100	app=pub	3d	Active
servidor	nsgrcp	ClusterIP	10.67.251.14	9080/TCP	-	app=servidor	4d	Active
sub	nspubsub	ClusterIP	10.67.241.218	8080/TCP	-	app=sub	3d	Active



Deployments — Filtered: 7 / 24 — Namespaces: nspubsub, nsp. — Search...

Name	Namespace	Pods	Replicas	Age	Conditions
dep1	nsproyecto	2/2	2	1w	Available Progressing
cliente	nsgrcp	1/1	1	5d	Available Progressing
grpcserver	nsgrcp	1/1	1	5d	Available Progressing
servidor	nsgrcp	1/1	1	5d	Available Progressing
apidummy	nsproyecto	1/1	1	3d	Available Progressing
pub	nspubsub	1/1	1	3d	Available Progressing
sub	nspubsub	1/1	1	3d	Available Progressing

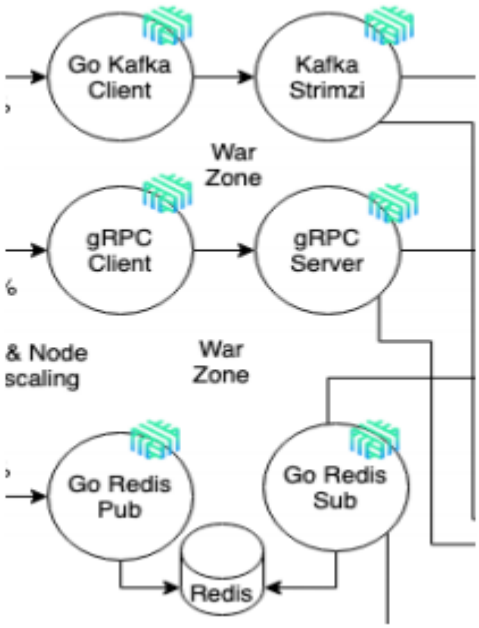
Pods Filtered: 9 / 32 Namespaces: nspubsub, nsp. Search...

Name	Namespace	Containers	Restarts	Controlled...	QoS	Age	Status
apidummy-567d496688-kgxj4	nsproyecto	■ ■ ■	0	ReplicaSet	Burstable	2d	Running
cliente-c9966dc9f-stzv9	nsgrcp	■ ■ ■	0	ReplicaSet	Burstable	1d	Running
dep1-76868c7f7b-6vmvx	nsproyecto	■ ■ ■	0	ReplicaSet	Burstable	2d	Running
dep1-76868c7f7b-6zspt	nsproyecto	■ ■ ■	0	ReplicaSet	Burstable	2d	Running
grpcserver-648bfd9c59-zl8gs	nsgrcp	■ ■ ■	0	ReplicaSet	Burstable	2d	Running
podproyecto	nsproyecto	■	0		BestEffort	1w	Running
pub-6b5d87468f-9c5dw	nspubsub	■ ■ ■	0	ReplicaSet	Burstable	2d	Running
servidor-84fcf49954-g4btr	nsgrcp	■ ■ ■	4	ReplicaSet	Burstable	1d	Running
sub-cf4697674-l76sh	nspubsub	■ ■ ■	7	ReplicaSet	Burstable	2d	Running

Tercera Parte:

gRPC

GRPC es un sistema de llamada a procedimiento remoto que procesa la comunicación en estructuras cliente-servidor distribuidas de manera especialmente eficiente gracias a una innovadora ingeniería de procesos.



Este sistema fue implementado en lenguaje GO, y consta de un cliente el cual es un servidor http que recibe las peticiones y se las envía a un servidor que escribe en la base de datos de MONGO. Para implementar fue necesario instalar las siguientes librerias:

```
"github.com/javiramos1/grpcapi"
"google.golang.org/grpc"
```

También se definió la estructura con la que se enviarán los datos de la siguiente manera:

```
type datos struct {
   Name         string `json:"name"`
   Location     string `json:"location"`
   Age          int    `json:"age"`
   InfectedType string `json:"infected_type"`
   State        string `json:"state"`
}
```

Una vez obtenida se le envia al servidor con la siguiente función:

```
func callService(c grpcapi.GrpcServiceClient, data []byte) (respuesta
error) {
   //  fmt.Println("callService...")
   req := &grpcapi.GrpcRequest{
       Input: string(data),
   }
   res, err := c.GrpcService(context.Background(), req)
   if err != nil {
       respuesta = err
       return
   }
   err = nil
   //muestra en consola los datos
   log.Printf("Response from Service: %v", res.Response)
   return
}
```

El servidor obtiene la información y la guarda en la base de datos MONGO, la función encargada de esto es la siguiente:

```

func mongoInsert(nombre string, departamento string, edad int, tipo
string, estado string) {
   clientOptions := options.Client().ApplyURI(uri)
   client, err := mongo.Connect(context.TODO(), clientOptions)
   if err != nil {
```

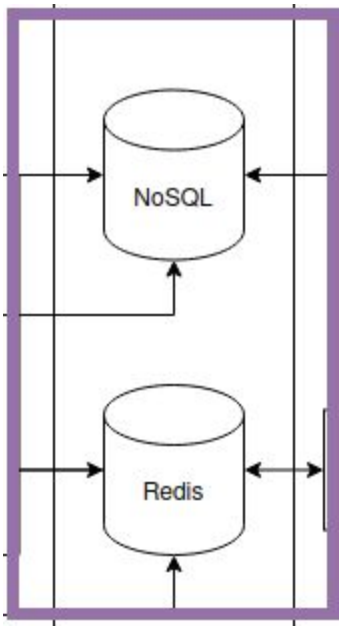
```
        log.Fatal(err)
    }
    err = client.Ping(context.TODO(), nil)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("Connected to MongoDB!")
    collection := client.Database("covid").Collection("covids")
    infectado := Infectado{nombre, departamento, edad, tipo, estado}
    fmt.Println(infectado)
    insertResult, err := collection.InsertOne(context.TODO(), infectado)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("Inserted a Single Document: ", insertResult.InsertedID)
}
```

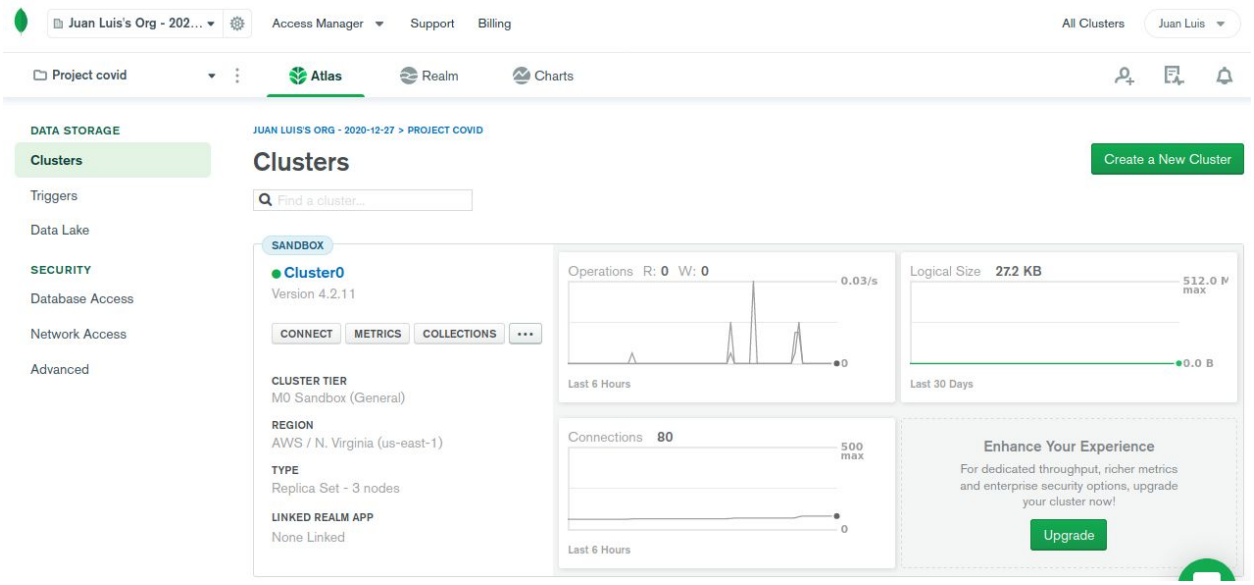
Cuarta Parte:

MONGODB

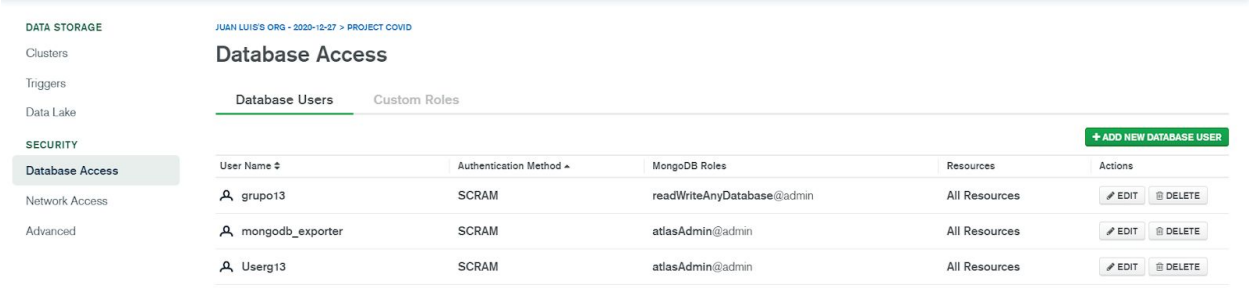
El cluster de MongoDB se implementó utilizando el servicio de MongoAtlas, el cual provee una capa gratuita.

El panel principal nos muestra el desempeño del cluster de mongo. El nombre de la base de datos es COVID y el de la colección COVIDS.

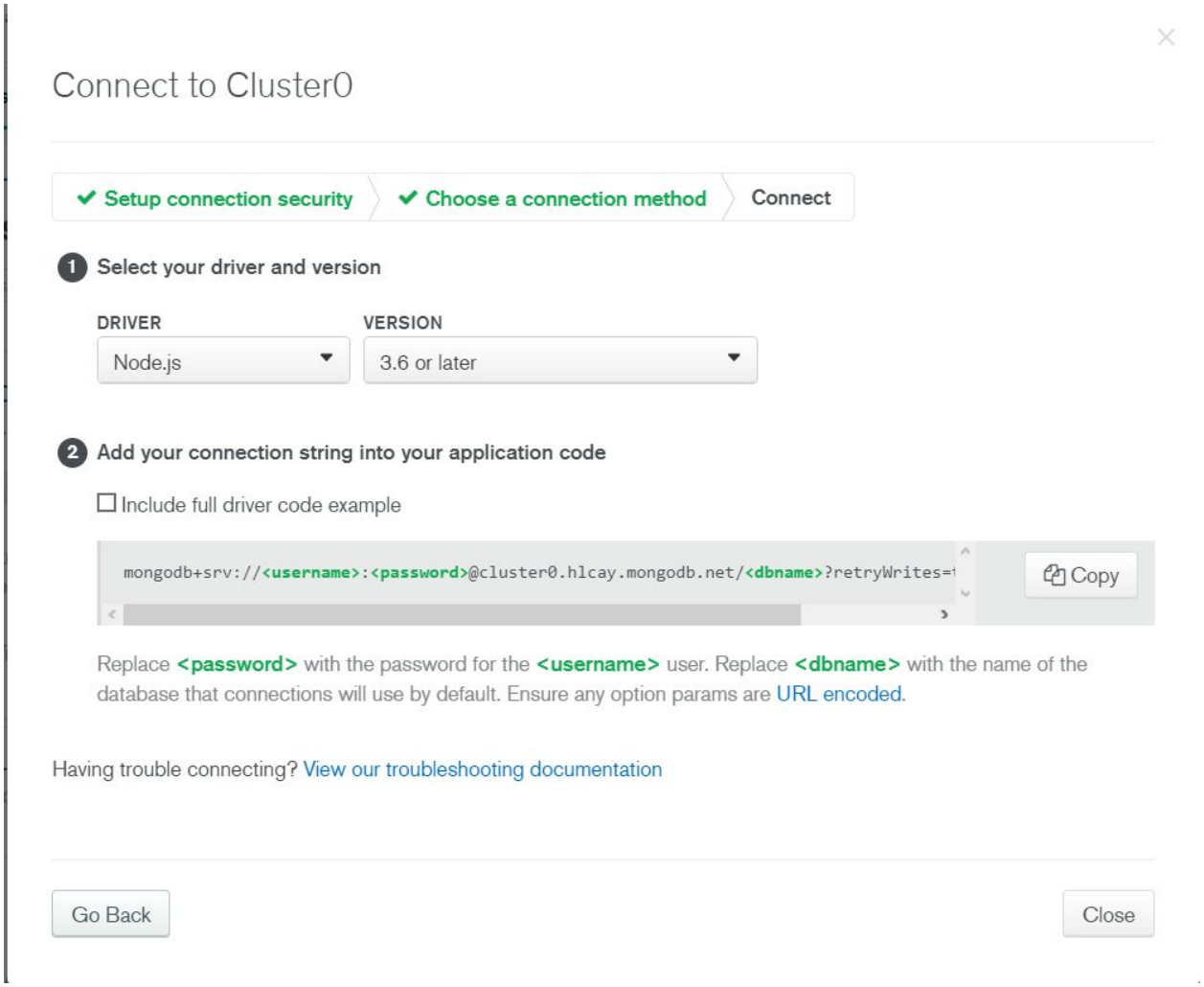




Para dar posteriormente a la creación del cluster se necesita lo que es la creación de diferentes usuarios con diferentes permisos y credenciales, esto para poder realizar las acciones de monitoreo y también de conexión para la obtención de datos, para ello se debe ingresar al apartado de ***Database Access*** lo cual nos mostrará la siguiente ventana la cual nos permitirá poder obtener una cadena de conexión o URI de la aplicación



Para obtener el ***URI*** se necesita ir a la pantalla inicial de clusters, luego de ello seleccionar el botón de conectar y escoger la opción de conectar aplicación lo cual nos mostrará lo siguiente junto con el URI de conexión.



REDIS

Primero para poder instalar lo que es REDIS necesitamos de una máquina virtual de ubuntu la cual nos servirá para poder correr esta base de datos nosql, para ello necesitaremos proceder a instalar por medio de los siguientes comandos

```
sudo apt update
sudo apt install redis-server
```

Esto nos servirá para instalar el server de redis, posteriormente ingresamos a su archivo conf y editamos dos partes, una necesaria para administrar las directivas init de redis y otra para configurar que este reciba el mismo puerto de la virtual.

```
# Note: these supervision methods only signal "process is ready."
#       They do not enable continuous liveness pings back to your supervisor.
supervised systemd

# If a pid file is specified, Redis writes it where specified at startup
# and removes it at exit.
```

```
# IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
# JUST COMMENT THE FOLLOWING LINE.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
bind 0.0.0.0 ::1

# Protected mode is a layer of security protection, in order to avoid that
# Redis instances left open on the internet are accessed and exploited.
#
```

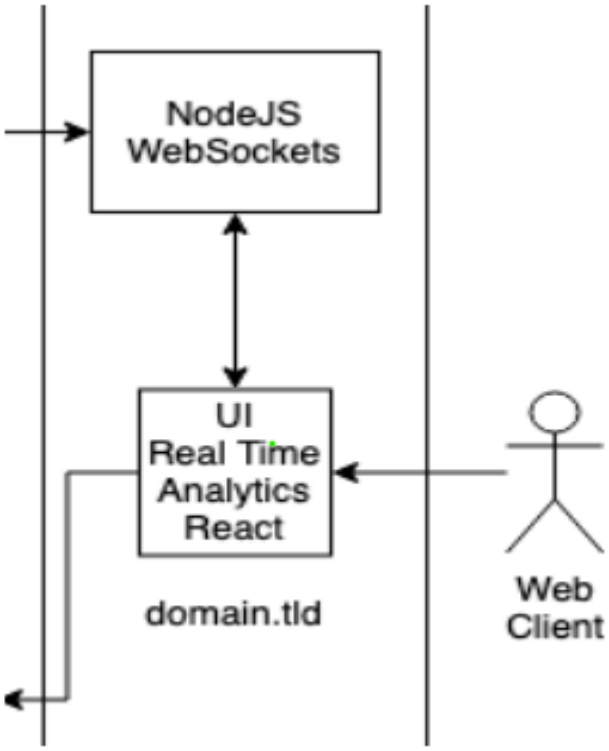
posteriormente podremos probar redis por medio del comando `redis-cli`

donde escribiremos ping y este nos responderá con PONG.

```
ubuntu@ip-172-31-36-19:~$ redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379>
```

Quinta Parte:

Para esta parte se está empleando lo que es un servidor de Node js el cual se sube a la nube, y una página web hecha con REACT la cual tendrá interacción constante con lo que es nuestro servidor de Node, para esto se realiza lo siguiente



Servidor de Node

```
const path=require('path');
const cors = require("cors")

const {mongoose} = require('./database');
// Settings

app.set('port',3100);

//middlewares

app.use(morgan('dev'));
app.use(express.json());
app.use(cors());
//routes
app.use('/api/datos',require('./routes/covidroutes'));



//static pages
app.use(express.static(path.join(__dirname,'public')))

app.listen(app.get('port'),()=>{
    console.log('server on port',app.get('port'));
})
```

Se crea un servidor de forma normal el cual tendrá una ruta de acceso para poder visualizar la información aquello almacenado en nuestra base de datos de MONGODB

```
router.get('/data',async (req,res)=>{
    const covidinfo = await covid.find();
    console.log(covidinfo);
    res.json(covidinfo)
});
```

Estos datos los obtiene por medio del comando *FIND* el cual nos devuelve todos los datos o bien aquellos filtrados por medio de diferentes parámetros que nosotros enviemos

Página web

Posteriormente para lo que es la creación de la página web la hacemos por medio de lo que es crear un componente donde nosotros pondremos la interfaz deseada para mostrar dentro de nuestras funciones, para ello usaremos el comando fetch el cual nos permite obtener datos de alguna dirección que nosotros indiquemos

```
async fetchingData(){
    await fetch('http://13.59.196.44:3100/api/datos/data')
    .then(res=>res.json())
    .then(data=>{
        this.setState({datas:data});
        //console.log(this.state.datas)
    })
    .catch(err=>console.log(err));
```

Posteriormente a obtener esta data pasamos a guardarla en el state o estado que es propio de react, esto con la finalidad de poder manipular los datos posteriormente.

Por último dentro del método render el cual nos permite a nosotros mostrar de forma gráfica aquellos datos obtenidos, según nosotros lo deseamos creamos un retorno, este retorno tendrá lo que es el código html necesario para mostrar la información

```
491         return (
492             <>
493             <div class="row">
494             <div class=" card text-white mb-3">
495             <div class="card-header"><h5>Todos los datos</h5></div>
496             <div class="card-body">
497                 <table class="table table-hover">
498                     <thead>
499                         <tr>
500                             <th>
501                                 NOMBRE
502                             </th>
503                             <th>
504                                 EDAD
505                             </th>
506                             <th>
507                                 DEPARTAMENTO
508                             </th>
509                             <th>
510                                 TIPO
511                             </th>
512                             <th>
513                                 ESTADO
```

Esto se hace mediante un mapeo de los datos que nosotros hemos obtenido, esto es el equivalente a un foreach de los datos, para ello se pone el arreglo que nosotros deseamos mapear y creamos un objeto que contendrá la información de cada objeto mapeado y así accederemos a sus diferentes datos hijos.