# www.covidero.ml

## Manual de Técnico


**Integrantes:**

Audrie Annelisse del Cid Ochoa 201801263

Lourdes Rosario Velásquez Melini 201906564

Romeo Ernesto Marroquín Sánchez 201902157

# Descripción de Herramientas

## Go

Es un lenguaje de programación concurrente y compilado inspirado en la sintaxis de C, que intenta ser dinámico como Python y con el rendimiento de C o C++. Ha sido desarrollado por Google y sus diseñadores iniciales fueron Robert Griesemer, Rob Pike y Ken Thompson.

Para este proyecto fue el lenguaje utilizado para la comunicación con las bases de datos no relacionales, y proporcionar a todo el despliegue una rápida implementación gracias a las ventajas que da utilizar Golang en los servidores dentro de contenedores en Kubernetes.

## Locust

Herramienta que se utiliza para la generación de tráfico hacia cualquier dirección en internet, en este caso, se utilizará una configuración específica de Locust para la realización de las peticiones correspondientes para realizar el ingreso a las bases de datos de la forma que corresponda.

El contenido de la petición estará definido por un archivo JSON el cual tiene numerosos registros de pacientes que fueron vacunados y que necesitan ser ingresados, el programa de Locust extrae la lista de pacientes y realiza envíos individuales de cada uno de los pacientes como peticiones diferentes, generando el tráfico correspondiente a la URL destino que se especifique.

## Docker

Es un proyecto de código abierto en el que se pueden realizar diversas acciones con contenedores, el cual se ha popularizado a nivel masivo, gracias a las facilidades que Docker da a sus contenedore es posible combinarlos con otras tecnologías con lo que puede automatizar el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción dentro de un sistema o modelo de virtualización de aplicaciones en múltiples sistemas operativos.

En este proyecto, esta herramienta permitió empaquetar las aplicaciones o servicios realizados en imágenes donde incluían todo lo necesario para la ejecución de cualquier aplicación dentro del contenedor.

## Google Compute Engine

Un servicio de computación seguro y personalizable con el que se puede crear y ejecutar instancias de máquinas virtuales, dentro de ellas se puede desplegar cualquier aplicación a través de su dirección IP pública la cual Google asigna a la instancia y puede ser accedida desde cualquier dispositivo que sea compatible con las reglas de firewall de dicha instancia.

## Google Kubernetes Engine

GKE es el primer servicio de Kubernetes totalmente gestionado que es compatible con todas las APIs de Kubernetes, el autoescalado en cuatro vías, los canales de lanzamiento y la gestión de varios clústeres, en este proyecto se utilizó un clúster de Kubernetes para desplegar toda la sección de procesamiento de las peticiones y gestión de base de datos, cada uno de los despliegues y los balanceadores utilizados dentro de la aplicación.

## Prometheus

Es un sistema que sirve para el monitoreo y el aviso de cualquier implementación, completamente personalizable, de código abierto, desarrollado por SoundCloud, normalmente utilizado para visualizar el rendimiento de aplicaciones en cualquiera de los ámbitos de desarrollo.

En este caso se utilizará para visualizar el estado de la máquina virtual que se encuentran las bases de datos, esto para visualizar los datos como tal tanto del tráfico como el estado general del dispositivo a través de objetos ilegibles los cuales pueden ser embellecidos utilizando Grafana.

## Grafana

Es un sistema que sirve únicamente para la apariencia visual de reportes que en un inicio tienen apariencias ilegibles, con espacios de trabajo completamente personalizables en los cuales se pueden colocar gráficas específicas las cuales se deseen mostrar, y los datos que se necesitan en ellas.

En este caso, se utilizó una plantilla ya creada de Grafana que se encuentran en bibliotecas, y utilizando el diseño establecido, se mostraron los datos que Prometheus había obtenido desde la máquina virtual donde corren las bases de datos de la aplicación.

## gRPC

Es un sistema de llamadas a procedimientos remotos de código abierto, desarrollado inicialmente por Google.

Aquí una aplicación cliente puede llamar directamente a un método en una aplicación de servidor en una máquina diferente como si fuera un objeto local. Facilitando la creación de aplicaciones y servicios distribuidos.

### *Servidor*

En el lado del servidor se define un servicio, el cual permite llamar métodos necesarios de forma remota con sus parámetros y tipos de retorno, que permiten manejar las llamadas de los clientes.

### *Cliente*

El lado del cliente cuenta con un código auxiliar que proporciona los mismos métodos que se mencionaron en el lado del servidor.

## Redis

Es una Base de Datos NoSQL que almacena la información mediante el formato de datos JSON. Además, los datos residen, principalmente, en memoria, lo que proporciona a este sistema unos muy buenos tiempos de respuesta en la recuperación de la información. Además de que es una forma de crear un sistema de cola, mensajería para aplicaciones nativas en la nube y arquitecturas de microservicios.

### *Pub*

En el código generamos una estructura la cual es:

```
type Persona struct {
    Name        string `json:"name"`
    Location    string `json:"location"`
    Age         int    `json:"age"`
    VaccineType string `json:"vaccine_type"`
    NDose       int    `json:"n_dose"`
}
```

Necesitamos crear nuestro cliente:

```
var redisClient = redis.NewClient(&redis.Options{
    Addr:     "bases.covidero.ml:6379",
    Password: "123",
    DB:       0,
})
```

A continuación, adquiriremos esos mismos datos del cuerpo usando la función c.BodyParser(), luego esos mismos datos tenemos que convertir en una cadena por eso usamos la función json.Marshall() :

```
func main() {
    app := fiber.New()

    app.Post("/", func(c *fiber.Ctx) error {
        p := new(Persona)

        if err := c.BodyParser(p); err != nil {
            panic(err)
        }

        payload, err := json.Marshal(p)
        if err != nil {
            panic(err)
        }

    if err := redisClient.Publish(ctx, "update", payload).Err(); err != nil {
        panic(err)
    }
    return c.SendStatus(200)
})
```

*Sub*

En el código generamos una estructura la cual es:

```
type Persona struct {
    Name        string `json:"name"`
    Location    string `json:"location"`
    Age         int    `json:"age"`
    VaccineType string `json:"vaccine_type"`
    NDose       int    `json:"n_dose"`
}
```

Necesitamos crear nuestro cliente:

```
var redisClient = redis.NewClient(&redis.Options{
    Addr:     "bases.covidero.ml:6379",
    Password: "123",
    DB:       0,
})
```

Para la conexión con la base de Mongo realizamos el siguiente código:

```go
func saveUser(usuario string) {
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
    defer cancel()

    mongoclient, err := mongo.Connect(ctx, options.Client().ApplyURI("mongodb://Uso1:123@bases.covidero.ml:27017/so1"))
    if err != nil {
        log.Fatal(err)
    }

    databases, err := mongoclient.ListDatabaseNames(ctx, bson.M{})
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(databases)

    Database := mongoclient.Database("so1")
    Collection := Database.Collection("patients")

    var bdoc interface{}

    errb := bson.UnmarshalExtJSON([]byte(usuario), true, &bdoc)

    fmt.Println(errb)

    insertResult, err := Collection.InsertOne(ctx, bdoc)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(insertResult)
}
```

Como ahora estamos trabajando en nuestro sub, estamos trabajando en nuestro receptor. De esta forma tenemos que crear un suscriptor usando la función redisClient.Subscribe(). Luego, creemos un bucle for para que imprima cada uno de los mensajes que recibimos a través de nuestro suscriptor. Dado que cada mensaje es una cadena, tendremos que volver a convertirlo a json. De esta forma usaremos la función json.Unmarshal()que tendrá dos argumentos, el primero será un búfer (una matriz de bytes a través del mensaje) y el segundo argumento es la estructura de persona.  Luego enviamos la información a la función saveUser() lo pasamos a string para que la función lo reciba.

```go
    subscriber := redisClient.Subscribe(ctx, "update")

    p := Persona{}

    for {
        msg, err := subscriber.ReceiveMessage(ctx)
        if err != nil {
            panic(err)
        }

        if err := json.Unmarshal([]byte(msg.Payload), &p); err != nil {
            panic(err)
        }

        fmt.Println("Received message from " + msg.Channel + " channel.")
        fmt.Printf("%+v\n", p)
        out, err := json.Marshal(p)
        if err != nil {
            panic (err)
        }
        saveUser(string(out))
    }
```