Tecnico

Facultad de ingeniería
LABORATORIO SISTEMAS OPERATIVOS 1
Sección A

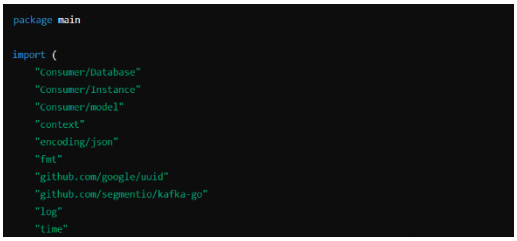
Por Grupo 4

ÍNDICE

Cosumer

tweets

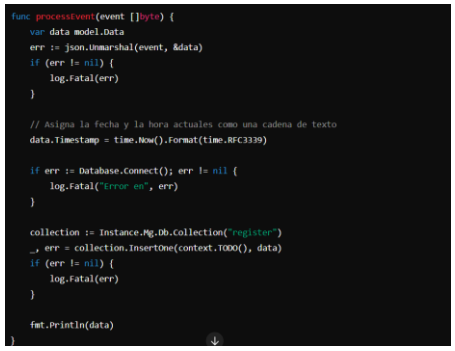
Este código está escrito en Go y es un consumidor de mensajes de Kafka. El consumidor lee mensajes de un tópico de Kafka, procesa los mensajes y los inserta en una base de datos MongoDB.



```go
package main

import (
    "Consumer/Database"
    "Consumer/Instance"
    "Consumer/model"
    "context"
    "encoding/json"
    "fmt"
    "github.com/google/uuid"
    "github.com/segmentio/kafka-go"
    "log"
    "time"
```

Este bloque importa los paquetes necesarios. Incluye paquetes estándar de Go, como context, encoding/json, fmt, log y time, así como paquetes externos como github.com/google/uuid y github.com/segmentio/kafka-go. También importa paquetes personalizados del proyecto Consumer.

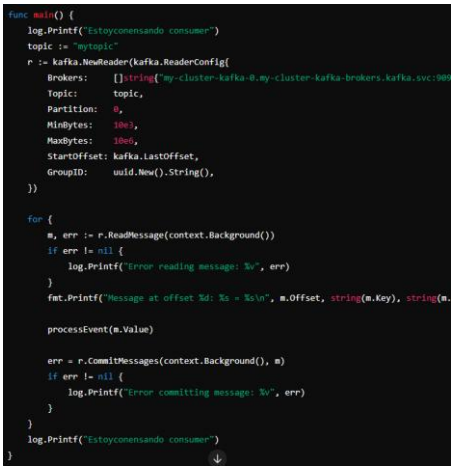
Función processEvent



```go
func processEvent(event []byte) {
    var data model.Data
    err := json.Unmarshal(event, &data)
    if (err != nil) {
        log.Fatal(err)
    }

    // Asigna la fecha y la hora actuales como una cadena de texto
    data.Timestamp = time.Now().Format(time.RFC3339)

    if err := Database.Connect(); err != nil {
        log.Fatal("Error en", err)
    }

    collection := Instance.Mg.Db.Collection("register")
    _, err = collection.InsertOne(context.TODO(), data)
    if (err != nil) {
        log.Fatal(err)
    }

    fmt.Println(data)
}
```

Esta función se encarga de procesar los eventos recibidos de Kafka:

1. Deserializa el mensaje JSON en una estructura model.Data.
2. Asigna una marca de tiempo actual al campo Timestamp.
3. Conecta a la base de datos MongoDB.
4. Inserta el evento en la colección register de MongoDB.
5. Imprime los datos en la consola.



```go
func main() {
    log.Printf("Estoyconensando consumer")
    topic := "mytopic"
    r := kafka.NewReader(kafka.ReaderConfig{
        Brokers:     []string{"my-cluster-kafka-0.my-cluster-kafka-brokers.kafka.svc:90
        Topic:       topic,
        Partition:   0,
        MinBytes:    10e3,
        MaxBytes:    10e6,
        StartOffset: kafka.LastOffset,
        GroupID:     uuid.New().String(),
    })

    for {
        m, err := r.ReadMessage(context.Background())
        if err != nil {
            log.Printf("Error reading message: %v", err)
        }
        fmt.Printf("Message at offset %d: %s = %s\n", m.Offset, string(m.Key), string(m.

        processEvent(m.Value)

        err = r.CommitMessages(context.Background(), m)
        if err != nil {
            log.Printf("Error committing message: %v", err)
        }
    }
    log.Printf("Estoyconensando consumer")
}
```

Esta es la función principal del programa:

1. Imprime un mensaje de inicio del consumidor.
2. Configura un lector de Kafka para leer mensajes del tópico mytopic.
3. En un bucle infinito:
4. Lee mensajes del tópico de Kafka.
5. Imprime el mensaje leído.
6. Procesa el mensaje llamando a processEvent.
7. Confirma que el mensaje ha sido procesado correctamente.

Explicación de los Componentes Clave

- Kafka Reader Config: Configura el lector de Kafka especificando los brokers, el tópico, l otros parámetros importantes.
- Deserialización JSON: Convierte el mensaje JSON en una estructura Go para su manipul
- Conexión a MongoDB: Conecta a una base de datos MongoDB y selecciona la colección
- Inserción de Datos: Inserta el mensaje procesado en la colección de MongoDB.

Kubernetes

consumer.yml

Propósito: Despliega y configura el consumidor de Kafka en Kubernetes.
Deployment: Define un conjunto de réplicas del pod del consumidor.
Service: Exponer el servicio del consumidor para que otros servicios o aplicaciones puedan comunicarse con él.

grafana.yml

Propósito: Despliega y configura Grafana en Kubernetes.
Deployment: Define un conjunto de réplicas del pod de Grafana.
Service: Exponer el servicio de Grafana para acceder a la interfaz web.
grpc.yml

Propósito: Despliega y configura un servicio gRPC en Kubernetes.
Deployment: Define un conjunto de réplicas del pod del servicio gRPC.
Service: Exponer el servicio gRPC para que otros servicios puedan comunicarse mediante el protocolo gRPC.

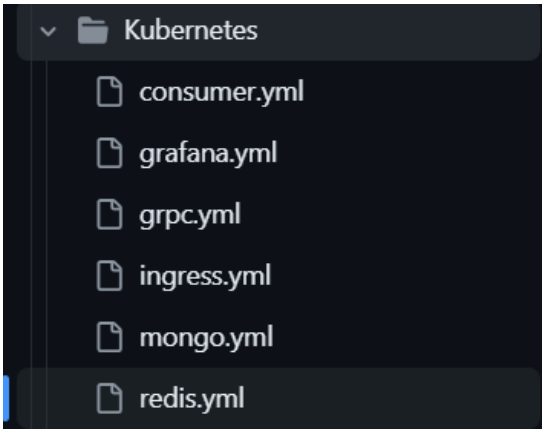
ingress.yml

Propósito: Configura el recurso Ingress en Kubernetes.
Ingress: Define reglas de enrutamiento HTTP y HTTPS a los servicios del clúster.

mongo.yml

Propósito: Despliega y configura MongoDB en Kubernetes.
StatefulSet: Define un conjunto de réplicas de MongoDB con volúmenes persistentes.
Service: Exponer el servicio de MongoDB para que otros servicios puedan conectarse a la base de datos.
redis.yml

Propósito: Despliega y configura Redis en Kubernetes.
StatefulSet o Deployment: Define un conjunto de réplicas de Redis.
Service: Exponer el servicio de Redis para que otros servicios puedan usarlo como una tienda de datos en memoria.



Locust

El código define un script de prueba de carga que simula tráfico de mensajes HTTP. Incluye dos tareas principales: enviar (POST) y recibir (GET) mensajes. Los mensajes se cargan desde un archivo `data.json`.

```python
import json
from random import randrange
from locust import HttpUser, between, task
```

json: Para trabajar con datos JSON.

randrange: Para seleccionar aleatoriamente elementos de una lista.

locust: Módulo principal de Locust para definir usuarios y tareas.

```python
class Reader():
    def __init__(self) -> None:
        self.array = []

    def pickRandom(self):
        length = len(self.array)
        if length > 0:
            random_index = randrange(0, length - 1) if length > 1 else 0
            return self.array.pop(random_index)
        else:
            print(">> Reader: No se encuentran valores en el archivo")
            return None

    def load(self):
        print(">> Reader: Iniciando lectura del archivo de datos")
        try:
            with open("data.json", "r") as data_file:
                self.array = json.loads(data_file.read())
        except Exception as error:
            print(f'>> Reader: Error en {error}')
```

- Reader: Clase que gestiona la lectura de datos desde un archivo JSON.
- pickRandom: Método que selecciona y elimina un elemento aleatorio de la lista.
- load: Método que carga datos desde `data.json` en un arreglo.

```python
class MessageTraffic(HttpUser):
    wait_time = between(0.1, 0.9)
    reader = Reader()
    reader.load()

    def on_start(self):
        print(">> MessageTraffic: Inicio de envío de tráfico")

    @task
    def PostMessage(self):
        random_data = self.reader.pickRandom()
        if random_data is not None:
            data_to_send = json.dumps(random_data)
            printDebug(data_to_send)
            self.client.post("/", json=random_data)
        else:
            print(">> MessageTraffic: Envío finalizado")
            self.stop(True)

    @task
    def GetMessage(self):
        self.client.get("/")
```

- **MessageTraffic:** Clase que define el comportamiento de un usuario en la prueba de carga.
- **wait_time:** Tiempo de espera entre tareas, entre 0.1 y 0.9 segundos.
- **reader:** Instancia de la clase `Reader` para gestionar los datos.
- **on_start:** Método que se ejecuta al iniciar el usuario, imprime un mensaje de inicio.
- **PostMessage:** Tarea que envía un mensaje POST con datos seleccionados aleatoriamente.
- **GetMessage:** Tarea que envía una solicitud GET a la raíz del servidor

Uso y Configuración

1. **Instalación de Dependencias:** Asegúrate de tener `Locust` instalado:

   ```
   bash
   Copiar código
   pip install locust
   ```

2. **Ejecución de la Prueba de Carga:** Puedes ejecutar la prueba de carga usando el siguiente comando en la terminal:

   ```
   bash
   Copiar código
   locust -f tu_archivo.py
   ```

   Luego, abre un navegador y navega a `http://localhost:8089` para iniciar la prueba y configurar el número de usuarios simulados.

Data Json Example


```json
[
    {
        "texto": "Calor sofocante",
        "pais": "Brasil"
    },
    {
        "texto": "Hace mucho viento hoy",
        "pais": "Argentina"
    },
    {
        "texto": "Agradable clima fresco",
        "pais": "Chile"
    },
    {
        "texto": "Lindo día soleado",
        "pais": "Australia"
    },
    {
        "texto": "Lindo día soleado",
        "pais": "Brasil"
    },
    {
        "texto": "Agradable clima fresco",
        "pais": "Australia"
    },
    {
        "texto": "Agradable clima fresco",
        "pais": "Brasil"
```

ClientGRPC

El código define un servidor web usando Fiber (un framework web para Go), que expone un endpoint `/insert`. Al recibir una solicitud POST en este endpoint, el servidor procesa los datos y los envía a dos servicios diferentes: uno gRPC y otro HTTP.

```go
import (
    "bytes"
    pb "clientGRPC/client"
    "context"
    "encoding/json"
    "fmt"
    "github.com/gofiber/fiber/v2"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"
    "io"
    "log"
    "net/http"
```

- bytes, encoding/json, fmt, io, log, net/http: Paquetes estándar de Go para manipulación de datos, registro, y comunicación HTTP.
- fiber/v2: Framework web para crear aplicaciones HTTP.
- grpc, grpc/credentials/insecure: Paquetes para implementar comunicación gRPC.
- context: Manejador de contexto en Go.
- pb: Paquete generado por Protobuf para el cliente gRPC.

```go
func sendToRust(data *Data) {
    jsonData, err := json.Marshal(data)
    if err != nil {
        log.Fatal(err)
    }

    res, err := http.Post("http://localhost:8000/set", "application/json", bytes.NewBuffer
    if err != nil {
        log.Fatal(err)
    }
    defer func(Body io.ReadCloser) {
        err := Body.Close()
        if err != nil {
            log.Fatal(err)
        }
    }(res.Body)

    if res.StatusCode != http.StatusOK {
        log.Fatalf("status code error: %d %s", res.StatusCode, res.Status)
    }
}
```

sendToRust: Función que envía los datos recibidos a un servicio HTTP en Rust.

- Convierte los datos a JSON.
- Realiza una solicitud POST al endpoint `http://localhost:8000/set`.
- Maneja errores y verifica el código de estado de la respuesta.

```go
func sendData(c *fiber.Ctx) error {
    var data map[string]string
    e := c.BodyParser(&data)
    if e != nil {
        return e
    }

    tweet := Data{
        Texto: data["texto"],
        Pais:  data["pais"],
    }

    go sendGrpcServer(tweet)
    go sendToRust(&tweet)

    return nil
}
```

sendData: Manejador para la ruta `/insert`.

- Parsea el cuerpo de la solicitud y lo convierte en un mapa de strings.
- Crea una instancia de `Data` con los valores recibidos.
- Llama a las funciones `sendGrpcServer` y `sendToRust` en goroutines para enviarlas a los servicios respectivos de manera concurrente.

```go
func sendGrpcServer(tweet Data) {
    conn, err := grpc.Dial("localhost:3001", grpc.WithTransportCredentials(insecure.NewCre
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }

    cl := pb.NewGetInfoClient(conn)
    defer func(conn *grpc.ClientConn) {
        err := conn.Close()
        if err != nil {
            log.Fatalf("could not close connection: %v", err)
        }
    }(conn)

    ret, err := cl.ReturnInfo(ctx, &pb.RequestId{
        Texto: tweet.Texto,
        Pais:  tweet.Pais,
    })
    if err != nil {
        log.Fatal(err)
    } else {
        fmt.Println("Respuesta del servidor ", ret)
    }
}
```

sendGrpcServer: Función que envía los datos recibidos a un servidor gRPC.

- Establece una conexión gRPC con `localhost:3001`.
- Crea un cliente gRPC usando el paquete generado por Protobuf.
- Envía una solicitud `ReturnInfo` con los datos recibidos.
- Maneja la respuesta y cierra la conexión.

redis-rust

Este Dockerfile consta de dos etapas: una etapa de construcción (`builder`) y una etapa de ejecución. Utiliza Rust para construir una aplicación y luego crea una imagen basada en Ubuntu para ejecutar la aplicación.

```
FROM rust:latest AS builder
WORKDIR /app
COPY . .
RUN cargo build --release
```

FROM rust: Esta línea define la imagen base para la etapa de construcción, que es la última versión de la imagen oficial de Rust. La etiqueta `builder` se usa para nombrar esta etapa, lo que permite referenciarla en etapas posteriores.

WORKDIR: Establece el directorio de trabajo dentro del contenedor en `/app`.

COPY :Copia todos los archivos del contexto de construcción (el directorio actual en el host) al directorio de trabajo del contenedor (`/app`).

RUN cargo build --release: Ejecuta el comando `cargo build --release` para compilar la aplicación Rust en modo de liberación. Los binarios compilados se almacenan en `/app/target/release/`.

- FROM ubuntu:22.04:
    - Define la imagen base para la etapa de ejecución, que es la versión 22.04 de Ubuntu.
- WORKDIR /app:
    - Establece el directorio de trabajo dentro del contenedor en `/app`.
- RUN mkdir -p /app:
    - Crea el directorio `/app` si no existe.
- COPY --from=builder /app/target/release/redis-rust /app/redis-rust:
    - Copia el binario compilado `redis-rust` desde la etapa `builder` al directorio de trabajo actual (`/app`) en la etapa de ejecución.
- COPY Rocket.toml /app/Rocket.toml:

    - Copia el archivo de configuración `Rocket.toml` del contexto de construcción al directorio de trabajo en el contenedor.

- RUN apt-get update && apt-get install -y libssl-dev ca-certificates && apt-get clean:

    - Actualiza la lista de paquetes y instala las dependencias necesarias (`libssl-dev` y `ca-certificates`) para ejecutar la aplicación.
    - Luego, limpia los archivos temporales de `apt-get` para reducir el tamaño de la imagen.

- CMD ["./redis-rust"]:

    - Especifica el comando que se ejecutará cuando se inicie un contenedor a partir de esta imagen. En este caso, ejecuta el binario `redis-rust`.

```
[package]
name = "redis-rust"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
rocket = { version = "0.5.0-rc.2", features = ["json"] }
redis = "0.23.0"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

serverGRPC

El código define un servidor gRPC en Go. Este servidor expone un método `ReturnInfo` que recibe datos de un cliente, los imprime y luego los envía a un productor de Kafka. Finalmente, el servidor responde al cliente con un mensaje de confirmación.

```go
import (
    "context"
    "fmt"
    "google.golang.org/grpc"
    "net"
    "serverGRPC/kafka"
    "serverGRPC/model"
    pb "serverGRPC/server"
)
```

- context: Proporciona el manejo de contexto para las solicitudes.
- fmt: Paquete de formato para entrada y salida.
- grpc: Paquete para implementar servidores y clientes gRPC.
- net: Proporciona funciones de red de bajo nivel.
- serverGRPC/kafka: Módulo personalizado para la interacción con Kafka.
- serverGRPC/model: Módulo personalizado que define el modelo de datos.
- pb "serverGRPC/server": Paquete generado por Protobuf que contiene las definiciones gRPC.

```go
func (s *server) ReturnInfo(ctx context.Context, in *pb.RequestId) (*pb.ReplyInfo, error) {
    tweet := model.Data{
        Texto: in.GetTexto(),
        Pais:  in.GetPais(),
    }

    fmt.Println(tweet)

    kafka.Produce(tweet)

    return &pb.ReplyInfo{Info: "Hola cliente, recibí el album"}, nil
```

ReturnInfo: Método que procesa las solicitudes del cliente.

- Recibe un contexto y un mensaje `RequestId` del cliente.
- Crea una instancia de `model.Data` a partir de los datos del cliente.
- Imprime los datos recibidos.
- Envía los datos a Kafka usando la función `Produce` del módulo `kafka`.
- Retorna una respuesta `ReplyInfo` al cliente.

main: Punto de entrada del programa.

- Crea un listener TCP que escucha en el puerto `3001`.
- Verifica si hay errores al crear el listener y detiene la ejecución en caso de error.
- Crea una nueva instancia de servidor gRPC.
- Registra el servidor gRPC usando `RegisterGetInfoServer`.
- Inicia el servidor gRPC para atender solicitudes entrantes en el puerto `3001`.

```go
func main() {
    listen, err := net.Listen("tcp", ":3001")
    if err != nil {
        panic(err)
    }
    s := grpc.NewServer()
    pb.RegisterGetInfoServer(s, &server{})

    if err := s.Serve(listen); err != nil {
        panic(err)
    }
}
```