

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERÍA
ESCUELA DE CIENCIAS Y SISTEMAS
ANÁLISIS Y DISEÑO DE SISTEMAS 1

USACTAR

Grupo : 13

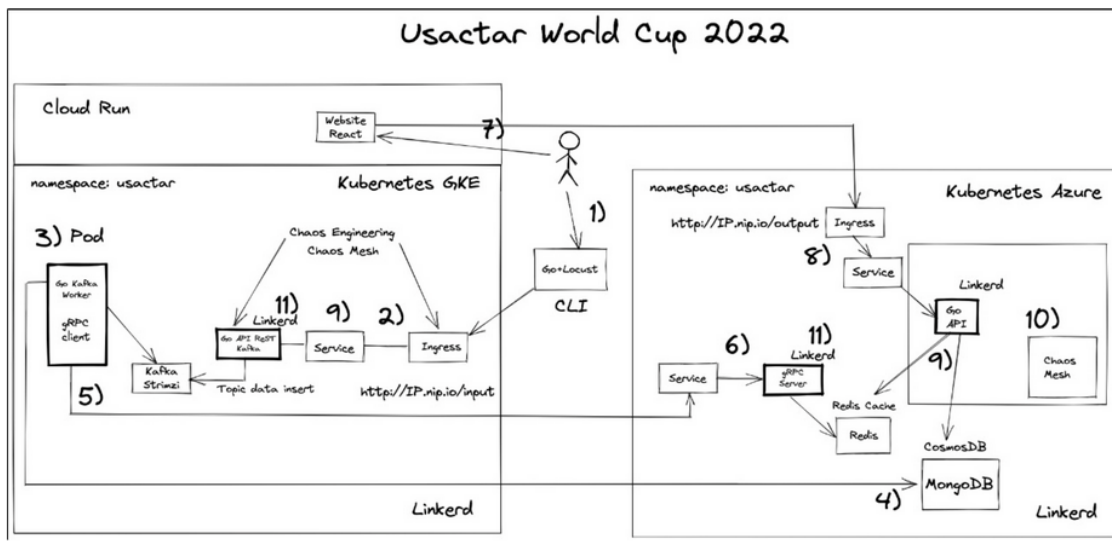
Nombre: Sara Paulina Medrano Cojulún Carné: 201908053

Nombre: Marvin Eduardo Catalán Véliz Carné: 201905554

Nombre: Julio José Orellana Ruíz Carné: 201908120

Fecha: 03/11/2022

Descripción General



Este proyecto consta de 2 clusters conectados por multicluster un cluster está hecho en Google Cloud Plataform y el otro en Azure.

El cluster de GCP consta con una api en GO que recibe todo el tráfico que nos genera un agente externo que es Locust, esta api transmite todo esto hacía una cola de kafka la cual está escuchando siempre, el worker de Kafka hace dos comunicaciones una que guarda datos en MongoDB y otro que manda tráfico hacia un gRPC cliente el cual transmite por medio de una estructura cliente-servidor hacia el servidor de gRPC este server se comunica directamente con una base de datos de Redis, con Redis Cache. Por último en el cluster de Azure hay una API a la cual se conecta el frontend creado en React, y montado en Cloud Run.

Conceptos generales:

DockerFile

Dockerfile es un archivo de texto que contiene las instrucciones necesarias para crear una nueva imagen del contenedor. Estas instrucciones incluyen la identificación de una imagen existente que se usará como base, los comandos que se ejecutarán durante el proceso de creación de la imagen y un comando que se ejecutará cuando se implementen instancias nuevas de la imagen del contenedor.

Cluster

Saber qué es un cluster consiste en entender que se trata de la conexión entre dos o más computadoras con el propósito de mejorar el rendimiento de los sistemas en la ejecución de diferentes tareas.

En el cluster, cada computadora se llama “nodo”, y no hay límites sobre cuántos nodos se pueden interconectar.

Con esto, las computadoras comienzan a actuar dentro de un solo sistema, trabajando juntas en el procesamiento, análisis e interpretación de datos e información, y/o realizando tareas simultáneas.

Deploy

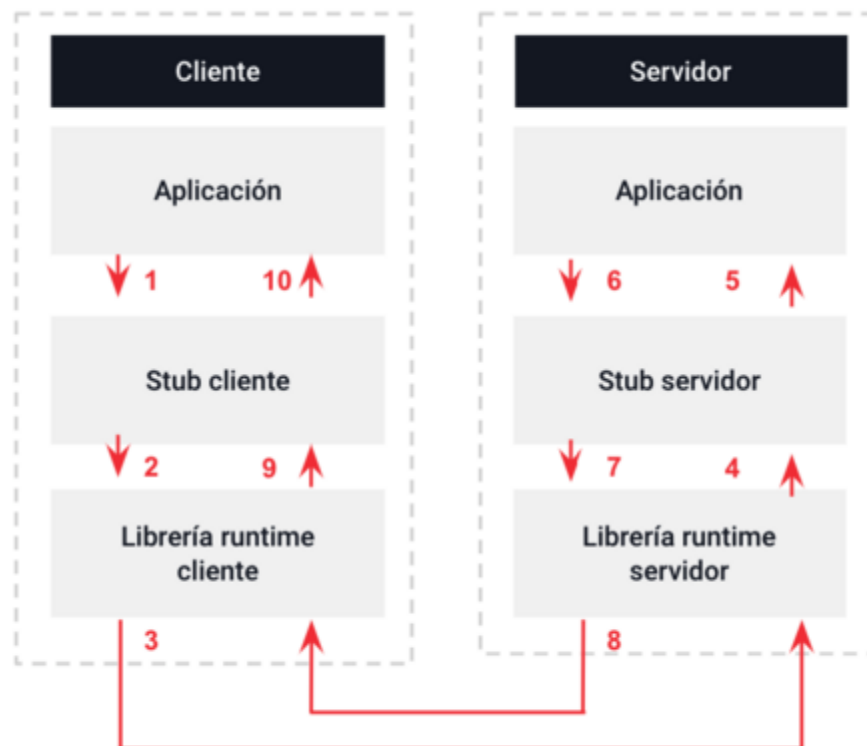
Deploy es un término famoso entre los desarrolladores web. Puede significar muchas cosas, dependiendo del ambiente y de la tecnología usada. Sin embargo, los significados que más se refieren a la práctica y pueden resumir su función son: implantar, colocar en posición, habilitar para uso o, simplemente, publicar.

GRPC

gRPC es un moderno 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. Actúa en el nivel del proceso, al igual que su antecesor, el sistema RPC. Un elemento característico de la comunicación entre procesos mediante gRPC es el principio de transparencia: la colaboración entre instancias (en parte muy) distanciadas es tan estrecha y sencilla que no se percibe ninguna diferencia en comparación con una comunicación local entre procesos internos de una máquina.

Desarrollado por Google en el año 2015, hoy es la Cloud Native Computing Foundation la encargada de su distribución y desarrollo. gRPC es un elemento de código abierto, es decir, el código fuente está accesible para que otros desarrolladores hagan modificaciones y participen en su desarrollo.

Por defecto, gRPC ejecuta el transporte de flujos de datos entre ordenadores alejados mediante **HTTP/2** y gestiona la estructura y la distribución de los datos mediante los Protocol buffers desarrollados por Google. Estos últimos se guardan en forma de archivos de texto plano con la extensión. *proto*.



GRPC - Cliente

Se basa en la estructura:



Proto: Es definir los mensajes que se van a intercambiar entre el cliente y el servidor. Posteriormente, tendremos que definir el servicio que utilizará dicho mensaje. Definiremos estos servicios y mensajes empleando una sintaxis de alto nivel, utilizando protocol buffers. Lo que haremos es crear un fichero con extensión **.proto** donde utilizaremos la sintaxis de protocol buffers.

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "io.grpc.examples.usactar";
option java_outer_classname = "usactarProto";
option objc_class_prefix = "HLW";

package usactar;

service GrpcConnection {
  rpc AddPrediction (PredictionRequest) returns (PredictionReply) {}
}

message PredictionRequest {
  string team1 = 1;
  string team2 = 2;
  string score = 3;
  int32 phase = 4;
}

message PredictionReply {
  string message = 1;
}
```

DockerFile

```
FROM node:latest
COPY ./ /app/
WORKDIR /app
RUN npm install
CMD ["node", "cliente.js"]
```

Cliente.Js

Es la clase donde se controla el gRPC client, aca accedemos a la configuración del proto y le damos la funcionalidad al gRPC

```
var PROTO_PATH = './proto/config.proto';

var parseArgs = require('minimist');
var grpc = require('@grpc/grpc-js');
var protoLoader = require('@grpc/proto-loader');
var packageDefinition = protoLoader.loadSync(
  PROTO_PATH,
  {keepCase: true,
    longs: String,
    enums: String,
    defaults: true,
    oneofs: true
  });
var usactar_proto = grpc.loadPackageDefinition(packageDefinition).usactar;

var argv = parseArgs(process.argv.slice(2), {
  string: 'target'
});
var target;

//VARIABLES API
const express = require('express');
var cors = require('cors');
const app = express();

app.use(express.json());
app.use(cors());
```

```

if (argv.target) {
    target = argv.target;
} else {
    target = '20.81.86.208:8083';
    //target = '0.0.0.0:50051';
}
var client = new usactar_proto.GrpcConnection(target,grpc.credentials.createInsecure());

app.post('/client-grcp',function (req,res){
    console.log(req.body);

    client.AddPrediction(req.body,function (err,response){
        res.status(200).json({mensaje: response.message})
    })
})

app.listen(3000,()=>{
    console.log('Servidor cliente en el puerto',3000);
})

```

GRPC – Servidor

Se basa en la estructura:



Proto: Es definir los mensajes que se van a intercambiar entre el cliente y el servidor. Posteriormente, tendremos que definir el servicio que utilizará dicho mensaje. Definiremos estos servicios y mensajes empleando una sintaxis de alto nivel, utilizando protocol buffers. Lo que haremos es crear un fichero con extensión **.proto** donde utilizaremos la sintaxis de protocol buffers.

```
message PredictionRequest {  
    string team1 = 1;  
    string team2 = 2;  
    string score = 3;  
    int32 phase = 4;  
}  
  
message PredictionReply {  
    string message = 1;  
}
```

DockerFile

```
FROM node:latest  
COPY ./ /app/  
WORKDIR /app  
RUN npm install  
CMD ["node", "servidor.js"]
```

Server.JS

servidor gRPC sobre la base de los Protocol Buffers. El cliente que hace la consulta, por su parte, genera el stub correspondiente. Si está disponible, la aplicación de cliente llama a la función correspondiente (“consulta de búsqueda de stock de un artículo”) en el stub.


```

var PROTO_PATH = './proto/config.proto';
const crypto = require('crypto');
var grpc = require('@grpc/grpc-js');
var protoLoader = require('@grpc/proto-loader');
var packageDefinition = protoLoader.loadSync(
  PROTO_PATH,
  {keepCase: true,
    longs: String,
    enums: String,
    defaults: true,
    oneofs: true
  });
var usactar_proto = grpc.loadPackageDefinition(packageDefinition).usactar;

const HOST_REDIS = 'azureCache-Redis.redis.cache.windows.net'
const KEY_REDIS = 'M63eFLchNx4pcX11OR6qJYsfjvX5wuumXAZCaMHhark='
const client = redis.createClient({
  //REDIS FOR TLS
  url: `rediss://${HOST_REDIS}:6380`,
  password: KEY_REDIS
});

async function AddPrediction(call, callback){
  let id = crypto.randomUUID();
  console.log(id)
  await client.set(id, JSON.stringify({
    team1: call.request.team1,
    team2: call.request.team2,
    score: call.request.score,
    phase: call.request.phase
  }))
  callback(null, {message: `Caso insertado en la base de datos ${id}`})
}

async function main(){

```

```

  await client.connect();
  var server = new grpc.Server();
  server.addService(usactar_proto.GrpcConnection.service, {
    AddPrediction: AddPrediction
  });
  server.bindAsync('0.0.0.0:50051', grpc.ServerCredentials.createInsecure(), () =>{
    server.start();
    console.log('gRPC server on port 50051')
  })
  console.log("\nEsperando trafico");

}

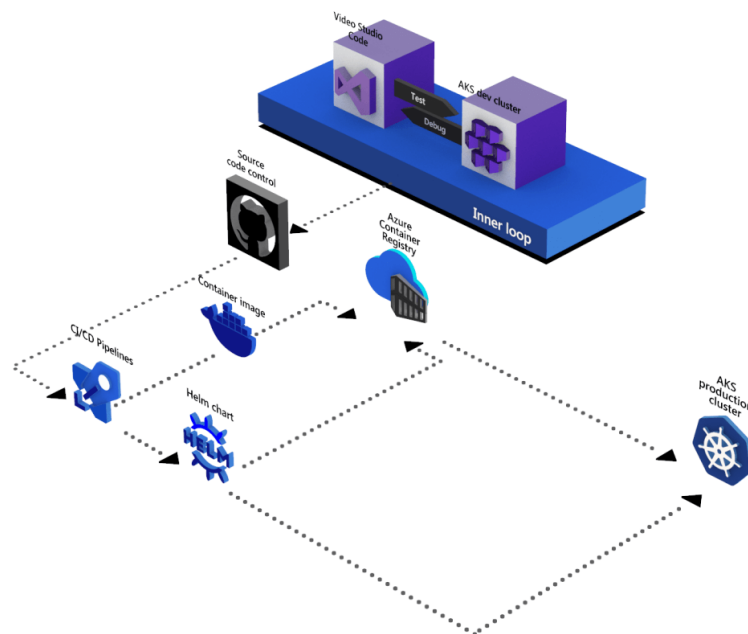
main();

```

Kubernetes

Kubernetes - AZURE

zure Kubernetes Service (AKS) ofrece la forma más rápida de empezar a desarrollar e implementar aplicaciones nativas de la nube en Azure, centros de datos o en el perímetro con canalizaciones de código a nube integradas y límites de protección. Obtenga administración y gobernanza unificadas para clústeres de Kubernetes locales, perimetrales y multinube. Interopere con los servicios de seguridad, identidad, administración de costos y migración de Azure.



Estructura:

```
deploy-go-api.yaml
deploy-grcp-server.yaml
deploy-namespace.yaml
```

Deploy-go-api.yaml

Nota: Es el punto de salida de AZURE consume REDIS y MOGO.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: deploy-go-api
    name: deploy-go-api
    namespace: space-azure
spec:
  replicas: 1
  selector:
    matchLabels:
      app: deploy-go-api
  template:
    metadata:
      labels:
        app: deploy-go-api
    spec:
      containers:
        - image: marved/go-api-azure
          name: go-kube
          imagePullPolicy: Always
          ports:
            - containerPort: 9093
---
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: svc-go-api
    name: svc-go-api
    namespace: space-azure
spec:
  type: LoadBalancer
  ports:
    - name: svc-go-api
      port: 9093
      targetPort: 9093
      protocol: TCP
  selector:
    app: deploy-go-api
```

deploy-grcp-server.yaml

Nota: Consume la imagen del servidor de GRPC.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: deploy-grcp-server
  name: deploy-grcp-server
  namespace: space-azure
spec:
  replicas: 1
  selector:
    matchLabels:
      app: deploy-grcp-server
  template:
    metadata:
      labels:
        app: deploy-grcp-server
    spec:
      containers:
        - image: marved/grpc-server
          name: node-kube
          imagePullPolicy: Always
          ports:
            - containerPort: 50051
---
```

```
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: svc-grcp-server
  name: svc-grcp-server
  namespace: space-azure
spec:
  type: LoadBalancer
  ports:
    - name: svc-grcp-server
      port: 8083
      targetPort: 50051
      protocol: TCP
  selector:
    app: deploy-grcp-server
---
```

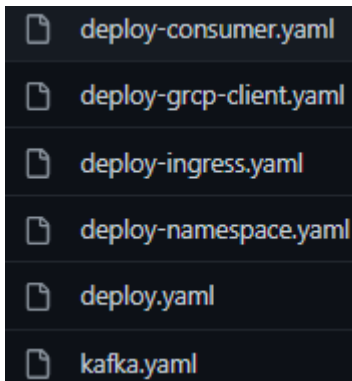
deploy-namespace.yaml

Nota: cluster virtual de AZURE, para esterilizarlo en los deploys anteriores y consuman un mismo espacio.

```
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: null
  name: space-azure
spec: {}
status: {}
```

Kubernetes – GCP

Estrcutura para GCP:



- deploy-consumer.yaml
- deploy-grcp-client.yaml
- deploy-ingress.yaml
- deploy-namespace.yaml
- deploy.yaml
- kafka.yaml

deploy-consumer.yaml

Nota: consume la imagen de Kafla por medio de GO.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: deploy-go-consumer-kafka
  name: deploy-go-consumer-kafka
  namespace: space-gcp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: deploy-go-consumer-kafka
  template:
    metadata:
      labels:
        app: deploy-go-consumer-kafka
    spec:
      containers:
        - image: marved/go-consumer-kafka:latest
          name: go-kube-consumer
          imagePullPolicy: Always
          ports:
            - containerPort: 9010
---

```

```

---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: svc-go-consumer-kafka
  name: svc-go-consumer-kafka
  namespace: space-gcp
spec:
  type: LoadBalancer
  ports:
    - name: svc-go-consumer-kafka
      port: 9010
      protocol: TCP
  selector:
    app: deploy-go-consumer-kafka

```

deploy-grcp-client.yaml

Nota: Consume el cliente del GRCP.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: deploy-grcp-client
  name: deploy-grcp-client
  namespace: space-gcp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: deploy-grcp-client
  template:
    metadata:
      labels:
        app: deploy-grcp-client
    spec:
      containers:
        - image: marved/grcp-client
          name: node-kube
          imagePullPolicy: Always
          ports:
            - containerPort: 3000
---

```

```

---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: svc-grcp-client
  name: svc-grcp-client
  namespace: space-gcp
spec:
  type: LoadBalancer
  ports:
    - name: svc-grcp-client
      port: 8083
      targetPort: 3000
      protocol: TCP
  selector:
    app: deploy-grcp-client
---

```

deploy-ingress.yaml

Nota: Consume el Ingress de nginx, con el servidor de nginx, aca se le da un protocolo HTTP y se da servicio el nginx con ingress.

```
apiVersion: v1
kind: Namespace
metadata:
  name: nginx-ingress
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nginx-ingress
  namespace: nginx-ingress
---
apiVersion: v1
kind: Secret
metadata:
  name: default-server-secret
  namespace: nginx-ingress
type: kubernetes.io/tls
data:
  tls.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSB0tLS0tck1JSUN2akNDQWZQ0NRREFPRjl0THNhW
  tls.key: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSU1FcEFJQkFBS0NBUEVBdi91R
```



```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
  namespace: nginx-ingress
data:
  ---
  #https://github.com/pablokbs/peladonerd/issues/19
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  creationTimestamp: null
  name: nginx-ingress-admin
  namespace: nginx-ingress
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: nginx-ingress
    namespace: nginx-ingress
  ---
  ---
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  name: nginx
  # annotations:
  #   ingressclass.kubernetes.io/is-default-class: "true"
spec:
  controller: nginx.org/ingress-controller
  ---
```

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-ingress
  namespace: nginx-ingress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-ingress
  template:
    metadata:
      labels:
        app: nginx-ingress
    spec:
      serviceAccountName: nginx-ingress
      containers:
        - image: nginx/nginx-ingress:edge
          imagePullPolicy: Always
          name: nginx-ingress
          ports:
            - name: http
              containerPort: 80
            - name: https
              containerPort: 443
          env:
            - name: POD_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: POD_NAME
              valueFrom:
                fieldRef:
```

```

        containerPort: 443
    env:
      - name: POD_NAMESPACE
        valueFrom:
          fieldRef:
            fieldPath: metadata.namespace
      - name: POD_NAME
        valueFrom:
          fieldRef:
            fieldPath: metadata.name
    args:
      - -nginx-configmaps=$(POD_NAMESPACE)/nginx-config
      - -default-server-tls-secret=$(POD_NAMESPACE)/default-server-secret
      - -enable-custom-resources=false
      #- -v=3 # Enables extensive logging. Useful for troubleshooting.
      #- -report-ingress-status
      #- -external-service=nginx-ingress
      #- -enable-leader-election
  ---
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
  namespace: nginx-ingress
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30000
      protocol: TCP
      name: http
  selector:
    app: nginx-ingress

```

deploy.yaml

Nota: despliega la aplicación y es consumida por go, en un puerto en específico.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: deploy-go
  name: deploy-go
  namespace: space-gcp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: deploy-go
  template:
    metadata:
      labels:
        app: deploy-go
    spec:
      containers:
        - image: marved/go-gcp:latest
          name: go-kube
          imagePullPolicy: Always
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: svc-go
  name: svc-go
  namespace: space-gcp
spec:
  type: LoadBalancer
  ports:

```

kafka.yaml

Nota: Se utiliza directamente el cluster de Kafka entre otras configuraciones.

```
#Creamos el cluster de kafka
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: kafka-cluster
  namespace: space-gcp
spec:
  kafka:
    version: 3.1.0
    replicas: 3
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
      - name: tls
        port: 9093
        type: internal
        tls: true
    config:
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 2
      log.message.format.version: "3.1"
      inter.broker.protocol.version: "3.1"
    storage:
      type: ephemeral
  zookeeper:
    replicas: 3
    storage:
      type: ephemeral
  entityOperator:
    topicOperator: {}
```

```

entityOperator:
  topicOperator: {}
  userOperator: {}
---
#Creamos el topic
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaTopic
metadata:
  name: input-kafka
  labels:
    strimzi.io/cluster: kafka-cluster
spec:
  partitions: 3
  replicas: 1
  config:
    retention.ms: 7200000
    segment.bytes: 1073741824

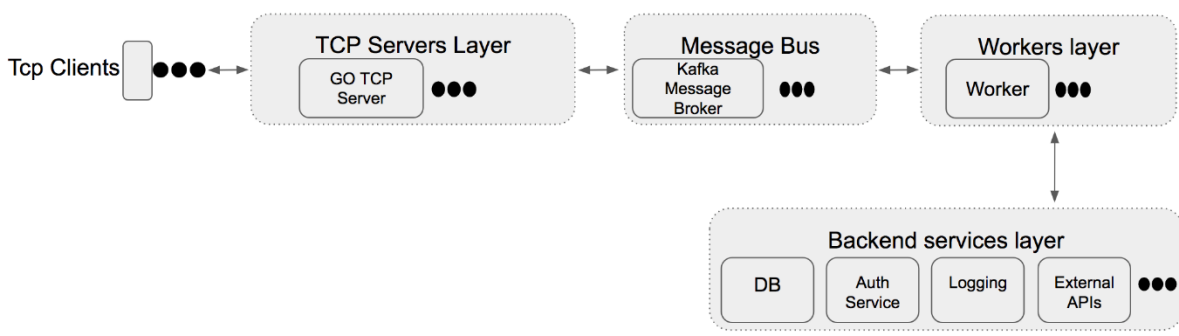
```

Go Consumer Kafka

Kafka está diseñado para soportar un alto volumen de mensajes pub-sub y streams. Kafka provee de un sistema de almacenamiento de mensajes de larga duración, similar a los logs, puedes ejecutarlo en una sola instancia, pero ha sido diseñado para ser distribuido con lo cual suele correr en un cluster, y almacena los registros en categorías llamadas topic.

Kafka almacena los registros dentro de diferentes particiones dentro de un mismo topic. Un topic, es una categoría sobre el cual los registros serán publicados, con lo cual **Kafka es capaz de escribir en diferentes colas que tengan el mismo topic.

Además Kafka nos permite decirle en cual de sus particiones queremos escribir dentro de un mismo topic. Imaginémonos una cajonera, donde la cajonera será nuestro topic y los cajones son nuestras particiones, nosotros podemos decidir en que parte vamos a guardar nuestra ropa ¿no?, pues igual pasa con los datos en Kafka.



Estructura Utilizada en el Proyecto.



DockerFile

```
FROM golang:1.18-alpine

WORKDIR /app

COPY go.mod ./
COPY go.sum ./
RUN go mod download

COPY . .

RUN go build -o /consumer

CMD ["/consumer"]
```

Main.go

Nota: Es el que consume Kafka y le da ejecuciones directamente con kafka. Esta parte esta representada con una red de consumo y contenido por métodos y funciones en su utilidad.

```
package main

import (
    "context"
    "fmt"
    "github.com/segmentio/kafka-go"
    "go.mongodb.org/mongo-driver/bson"
    "go.mongodb.org/mongo-driver/mongo"
    "go.mongodb.org/mongo-driver/mongo/options"
    "io/ioutil"
    "log"
    "net/http"
    "strings"
    "time"
)

var MONGO_URL = "mongodb://cosmosdb-mongo-sopes1:FQZmTPpDpwOjpqkAUdkdLrT8qnCZktOjbm3Fi6R2lddZU

func saveGrpc(dato string) {
    url := "http://34.66.132.45:8083/client-grpc"
    method := "POST"

    payload := strings.NewReader(dato)

    client := &http.Client{}
    req, err := http.NewRequest(method, url, payload)
```



```

func main() {
    log.Println("Estoy en el consumer :)")
    conf := kafka.ReaderConfig{
        Brokers:      []string{"kafka-cluster-kafka-bootstrap:9092"},
        Topic:         "input-kafka",
        GroupID:       "g1",
        StartOffset:  kafka.LastOffset,
    }
    reader := kafka.NewReader(conf)

    for {
        message, err := reader.ReadMessage(context.Background())
        if err != nil {
            fmt.Println("Ocurrio un error", err)
            continue
        }
        fmt.Println("El mensaje es: ", string(message.Value))
        saveGrpc(string(message.Value))
        var doc interface{}
        errr := bson.UnmarshalExtJSON([]byte(message.Value), true, &doc)
        if errr != nil {
            log.Fatal(errr)
        }
        //CONEXION A LA BASE DE DATOS E INSERCIÓN DE DATOS
        client, err := mongo.NewClient(options.Client().ApplyURI(MONGO_URL))
        if err != nil {
            log.Fatal(err)
        }
        ctx, _ := context.WithTimeout(context.Background(), 10*time.Second)
        err = client.Connect(ctx)
        if err != nil {

```

```

message, err := reader.ReadMessage(context.Background())
if err != nil {
    fmt.Println("Ocurrio un error", err)
    continue
}
fmt.Println("El mensaje es: ", string(message.Value))
saveGrpc(string(message.Value))
var doc interface{}
errr := bson.UnmarshalExtJSON([]byte(message.Value), true, &doc)
if errr != nil {
    log.Fatal(errr)
}
//CONEXION A LA BASE DE DATOS E INSERCIÓN DE DATOS
client, err := mongo.NewClient(options.Client().ApplyURI(MONGO_URL))
if err != nil {
    log.Fatal(err)
}
ctx, _ := context.WithTimeout(context.Background(), 10*time.Second)
err = client.Connect(ctx)
if err != nil {
    log.Fatal(err)
}
defer client.Disconnect(ctx)

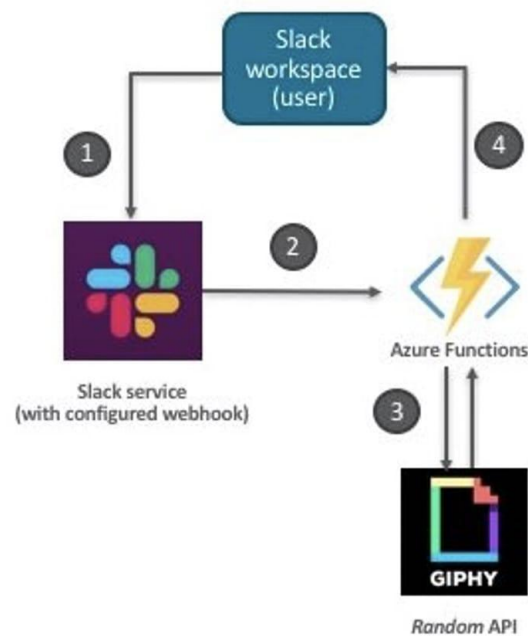
collection := client.Database("UsactarMongoDB").Collection("data")
res, insertErr := collection.InsertOne(ctx, doc)
if insertErr != nil {
    log.Fatal(insertErr)
}
fmt.Println(res)
}

```

GO API Azure

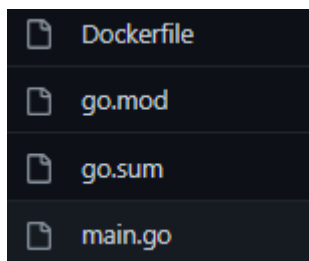
Azure SDK para Go proporciona varias bibliotecas (agrupadas en administración y cliente) que permiten que el código de Go se comunique con los servicios de Azure. Tanto la administración como las bibliotecas cliente están diseñadas para trabajar con entornos locales y en la nube.

Debido a la adopción de genéricos, El SDK de Azure para Go es compatible con Go 1.18 y versiones posteriores. En el futuro, el SDK de Azure para Go admitirá las dos versiones principales más recientes



Las bibliotecas de administración y cliente se basan en la API REST de Azure. Esta jerarquía le permite acceder a la funcionalidad de la API REST de Azure desde el léxico de Go con el que está familiarizado. También puede usar la API REST de Azure directamente desde el código de Go.

Estructura Utilizada en el proyecto:



DockerFile:

```
FROM golang:1.18-alpine

WORKDIR /app

COPY go.mod ./
COPY go.sum ./
RUN go mod download

COPY . .

RUN go build -o /consumer

CMD ["/consumer"]
```

Main.go

Nota: Es el que consume Azure por medio de la api construida en GO y le da ejecuciones directamente con Azure. Esta parte esta representada con una red de consumo y contenido por métodos y funciones en su utilidad.

```
type prediction struct {
    Team1 string `json:"team1"`
    Team2 string `json:"team2"`
    Score string `json:"score"`
    Phase int    `json:"phase"`
}

var MONGO_URL = "mongodb://cosmosdb-mongo-sopes1:FQZmTPpDpwOjppqkAUdkdLrT8qnCZkt0jbm3Fi6R2lddZUS

/*
 *
Funcion para iniciar el servidor y ver que conecte
 */
func Inicio(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Conexión exitosa api Go")
    log.Println("Bienvenido API")
}

func getAllMongo(w http.ResponseWriter, r *http.Request) {
    //CONEXION A LA BASE DE DATOS E INSERCIÓN DE DATOS
    client, err := mongo.NewClient(options.Client().ApplyURI(MONGO_URL))
    if err != nil {
        log.Fatal(err)
    }
    ctx, _ := context.WithTimeout(context.Background(), 10*time.Second)
    err = client.Connect(ctx)
    if err != nil {
        log.Fatal(err)
    }
    defer client.Disconnect(ctx)

    collection := client.Database("UsactarMongoDB").Collection("data")
```

```

collection := client.Database("UsactarMongoDB").Collection("data")

filter := bson.D{{}}

cursor, err := collection.Find(context.TODO(), filter)
if err != nil {
    panic(err)
}

var results []prediction
if err = cursor.All(context.TODO(), &results); err != nil {
    panic(err)
}
w.Header().Set("Content-Type", "application/json")
w.WriteHeader(http.StatusOK)
json.NewEncoder(w).Encode(results)
}

```

```

func getAllRedis(w http.ResponseWriter, r *http.Request) {
    client := redis.NewClient(&redis.Options{
        Addr:      "azureCache-Redis.redis.cache.windows.net:6380",
        Password:  "hADq0GGJgiULQECI2u1jrwRFfad9a3KgMeAzCaNfEQAU=",
        TLSConfig: &tls.Config{MinVersion: tls.VersionTLS12},
    })
    // test connection
    pong, err := client.Ping().Result()
    if err != nil {
        log.Fatal(err)
    }

    iter := client.Scan(0, "prefix:*", 1000).Iterator()
    for iter.Next() {
        fmt.Println("keys", iter.Val())
    }
    if err := iter.Err(); err != nil {
        fmt.Println("test error")
        panic(err)
    }
    // return pong if server is online
    //fmt.Println(pong)
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(pong)
}

func main() {
    router := mux.NewRouter().StrictSlash(true)
    router.HandleFunc("/", Inicio).Methods("GET")
    router.HandleFunc("/get-all-mongo", getAllMongo).Methods("GET")
    router.HandleFunc("/get-all-redis", getAllRedis).Methods("GET")
    log.Println("Server iniciado en el puerto 9093")
    log.Fatal(http.ListenAndServe(":9093", handlers.CORS(handlers.AllowedHeaders([]string{"X-Request

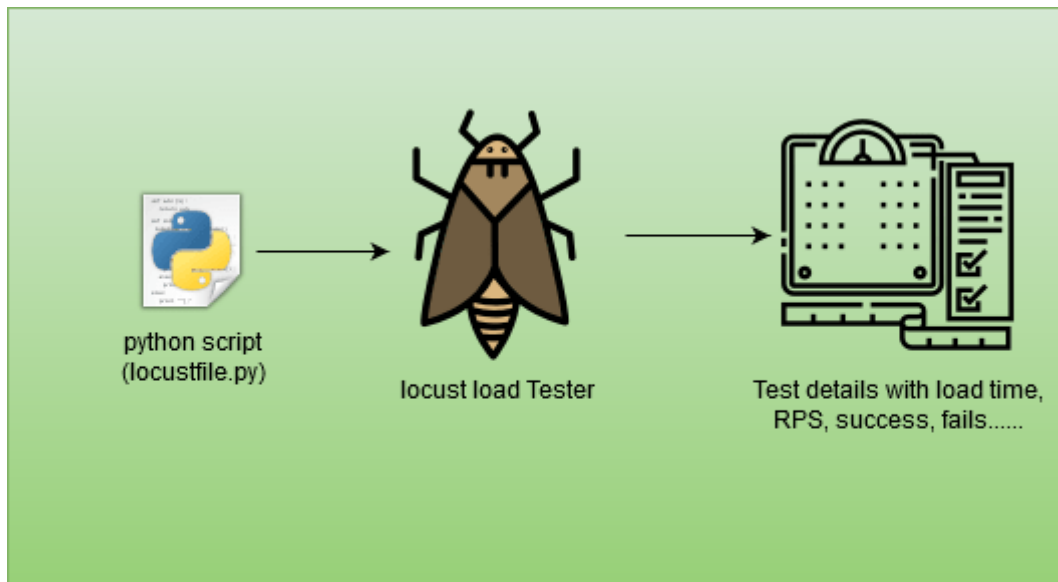
```

GO + LOCUST

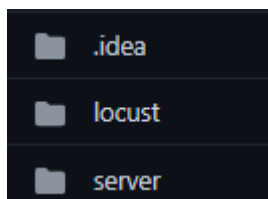
go-locust es una biblioteca para controlar la generación de carga de langostas y obtener estadísticas escritas en golang. Este es el cliente que permite iniciar, detener una prueba de carga de langosta y aumentar la carga. Esto utiliza puntos finales de Locust para comunicarse con Locust.

Actualmente, go-locust requiere Go versión 1.13 o superior y Locust 0.14 o superior. Haré todo lo posible para probar con versiones anteriores de Go y Locust, pero debido a limitaciones de tiempo, no he probado con versiones anteriores.

Esto no procesa las estadísticas del cliente y presenta la información tal como es.



Estrucutra del proyecto utilizada:



LOCUST

traffic.json

traffic.py

Traffic.json

Es un ejemplo de los datos, o de como se van a interpretar.

```
[
  {
    "team1": "Guatemala",
    "team2": "Argentina",
    "score": "2-5",
    "phase": 1
  },
  {
    "team1": "Nicaragua",
    "team2": "Panama",
    "score": "6-7",
    "phase": 1
  },
  {
    "team1": "Argentina",
    "team2": "Colombia",
    "score": "0-0",
    "phase": 3
  }
]
```

Traffic.py

Nota: Esta clase se utiliza para la lectura de trafico o bien la lectura de los datos. En esta parte accedemos a los datos, los leemos y los interpretamos según los requerimientos.

```

import json
from random import random, randrange
from sys import getsizeof
from locust import HttpUser, task, between
#librerias que usa el sistema
debug = True

def printDebug(msg): # metodo para mostrar mensajes el debug
    if debug:
        print(msg)

def loadData(): #metodo para cargar la informacion
    try:
        with open("traffic.json", 'r') as data_file: #buscamos y recorremos el archivo
            array = json.loads(data_file.read())
            return array
        print (f'>> Reader: Datos cargados correctamente, {len(array)} datos -> {getsizeof(array)} bytes.')
    except Exception as e: # si ocurre una excepcion
        print (f'>> Reader: No se cargaron los datos {e}')
        return []

array = loadData()

```

```

class Reader(): #clase para leer el archivo
    def pickRandom(self): #para poder acceder de manera aleatoria a los datos
        length = len(array)
        if (length > 0):
            random_index = randrange(0, length - 1) if length > 1 else 0
            return array.pop(random_index) #luego de tomar el valor del array, lo libera para ya no ser tomado en cuenta
        else:
            print (">> Reader: No hay más valores para leer en el archivo.")
            return None #finaliza la lectura del archivo

class MessageTraffic(HttpUser):
    wait_time = between(0.1, 0.9) #intervalo entre cada peticion
    def on_start(self):
        print (">> MessageTraffic: Iniciando el envio de tráfico") #mensaje para definir el inicio del trafico
        self.reader = Reader()

    @task
    def PostMessage(self):
        random_data = self.reader.pickRandom() #tomamos el valor random
        if (random_data is not None):
            data_to_send = json.dumps(random_data) #leemos el archivo
            printDebug (data_to_send)
            self.client.post("/input", json=random_data) #endpoint que se va a consumir
        else:
            print(">> MessageTraffic: Envio de tráfico finalizado, no hay más datos que enviar.")
            self.stop(True)

```

Server GO y Lotus

Estructura utilizada en el proyecto



DockerFile:

```
FROM golang:1.17 as build-env

WORKDIR /go/src/app
COPY *.go .

RUN go mod init
RUN go get -d -v ./...
RUN go vet -v
RUN go test -v

RUN CGO_ENABLED=0 go build -o /go/bin/app

CMD ["/go/bin/app"]
```

Main.go

```

import (
    "context"
    "encoding/json"
    "fmt"
    "github.com/gorilla/handlers"
    "github.com/gorilla/mux"
    "github.com/segmentio/kafka-go"
    "io/ioutil"
    "log"
    "net/http"
    "strconv"
    "time"
)

type prediction struct {
    Team1 string `json:"team1"`
    Team2 string `json:"team2"`
    Score string `json:"score"`
    Phase int    `json:"phase"`
}

/*
 *
 Funcion para iniciar el servidor y ver que conecte
 */
func Inicio(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Conexión exitosa api Go")
    log.Println("Petición de inicio")
}

```

```

func newPrediction(w http.ResponseWriter, r *http.Request) {
    body, err := ioutil.ReadAll(r.Body)
    if err != nil {
        panic(err)
    }
    var newpred prediction

    err = json.Unmarshal(body, &newpred)

    if err != nil {
        panic(err)
    }

    var jsonPrediction = string(`{"team1":` + newpred.Team1 + `, "team2":` + newpred.Team2 + `, "score":` + newpred.Score + `, "phase":` + newpred.Phase + `}`)
    conn, _ := kafka.DialLeader(context.Background(), "tcp", "kafka-cluster-kafka-bootstrap:9092", "input-kafka", 0)
    conn.SetWriteDeadline(time.Now().Add(time.Second * 10))
    conn.WriteMessages(kafka.Message{Value: []byte(jsonPrediction)})
    log.Println(jsonPrediction)
    w.Write([]byte(jsonPrediction))
}

func main() {
    router := mux.NewRouter().StrictSlash(true)
    router.HandleFunc("/inicio", Inicio).Methods("GET")
    router.HandleFunc("/input", newPrediction).Methods("POST")
    log.Println("Server iniciado en el puerto 80")
    log.Fatal(http.ListenAndServe(":80", handlers.CORS(handlers.AllowedHeaders([]string{"X-Requested-With", "Content-Type"})).Handler(router)))
}

```

Front-end

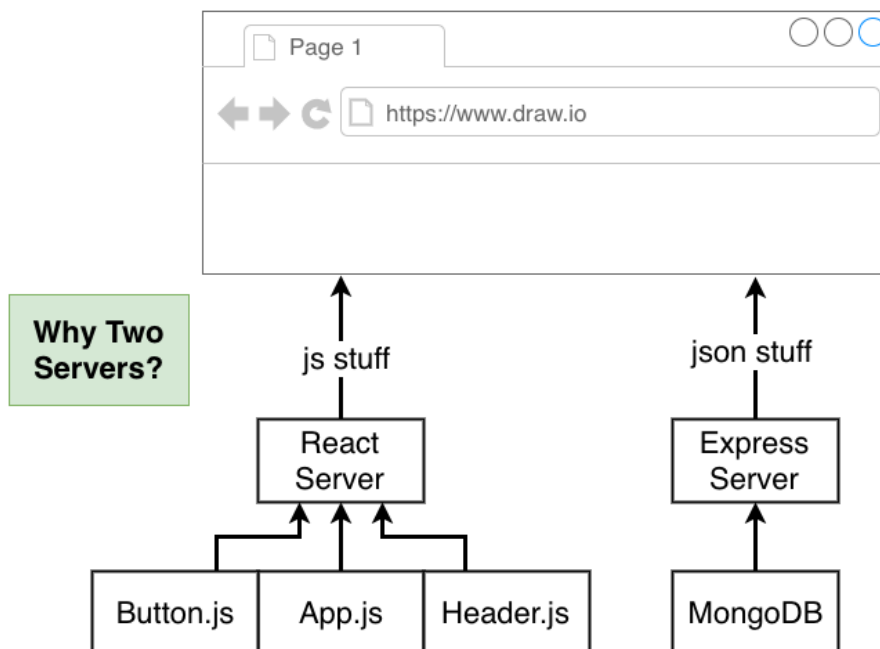
React-JS

React es una biblioteca o librería de código abierto que está escrita en JavaScript. Fue desarrollada por Facebook en el 2013 con la finalidad de facilitar la creación de componentes reutilizables e interactivos para las interfaces de usuario.

Uno de sus puntos más destacados es que no solo se usa en el lado del cliente, sino que también se puede representar en el servidor y trabajar juntos.

Son datos interesantes que React se utiliza en Facebook para la producción de componentes y que Instagram está escrito enteramente en React. Adicionalmente, también se utiliza en otras plataformas como Netflix, PayPal, AirBnb, Uber, Reddit y Twitter.

React es una excelente alternativa para realizar todo tipo de aplicaciones web o para dispositivos móviles, así como para crear single page applications (SPA o aplicaciones de una sola página).



¿Cómo funciona React JS?

Para comprender cómo funciona React es clave que nos situemos en un contexto, pues cuando se aprende desarrollo web se obtiene conocimiento de tres conceptos básicos:

- HTML: la semántica, estructura e información de la página web; es decir, su esqueleto.
- CSS: la apariencia de nuestra página web.
- JavaScript: básicamente es el cerebro de nuestra página. Determina qué hacer en función de lo que sucede en ella.

Sin embargo, antes de React estos conceptos funcionaban por separado, en diferentes archivos y carpetas, por lo que escalar y extraer diversas partes del código para reutilizar o migrar funcionalidades era más complicado.

Por esto, los ingenieros de Facebook decidieron incluir todo en un solo paquete, al que llamaron “componente”. En los componentes, la estructura HTML y JS son inseparables, y combinables con CSS.

Esto hizo posible la implementación de una nueva notación que hace más eficiente el desarrollo de aplicaciones escalables: JSX, que significa JavaScript XML.

Estructura del Font-end

public	Frontend	4 days ago
src	Frontend	4 days ago
.env	Frontend	4 days ago
.eslintrc.js	Frontend	4 days ago
.gitignore	Frontend	4 days ago
Dockerfile	update	29 minutes ago
craco.config.js	Frontend	4 days ago
nginx.conf	agregamos nginx conf	2 hours ago
package-lock.json	update	35 minutes ago
package.json	update	35 minutes ago
yarn.lock	update	35 minutes ago

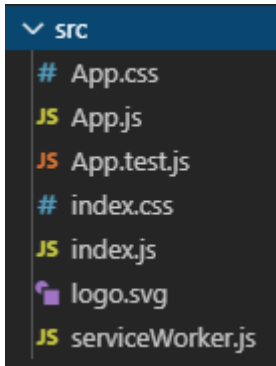
Carpeta public:

Contiene los archivos (estáticos, los que no cambian) que nos va a permitir montar la aplicación. Si nos fijamos en el interior de esta carpeta, tenemos el archivo index.html que es un HTML de toda la vida. Y un conjunto de ficheros que van relacionados con index.html.

En este caso en particular, al utilizar el comando create-react-app el div tiene el id root aunque podemos darle el nombre que deseemos sin problema alguno. Eso sí, es importante tener detectado el nombre ya que le atacaremos desde algún fichero del directorio src (lo vemos más abajo).

Carpeta src(Source):

La carpeta src, es la carpeta sobre la que se encuentra nuestro código de React, y por tanto, la carpeta donde trabajaremos.



Distribucion /SRC

📁 @core	Frontend	4 days ago
📁 assets	Frontend	4 days ago
📁 configs	Frontend	4 days ago
📁 layouts	Frontend	4 days ago
📁 navigation	Frontend	4 days ago
📁 redux	Frontend	4 days ago
📁 router	Frontend	4 days ago
📁 utility	Frontend	4 days ago
📁 views	Frontend	4 days ago
📄 App.js	Frontend	4 days ago
📄 App.test.js	Frontend	4 days ago
📄 index.js	Frontend	4 days ago
📄 index.scss	Frontend	4 days ago
📄 serviceWorker.js	Frontend	4 days ago

Index.js

Si abrimos el fichero index.js. Vamos a analizar el contenido del fichero:

```

JS index.js  X
src > JS index.js
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import './index.css';
4  import App from './App';
5  import * as serviceWorker from './serviceWorker';
6
7  ReactDOM.render(<App />, document.getElementById('root'));
8
9  // If you want your app to work offline and load faster, you can change
10 // unregister() to register() below. Note this comes with some pitfalls.
11 // Learn more about service workers: https://bit.ly/CRA-PWA
12 serviceWorker.unregister();

```

- Línea 1: importa el módulo de React que como ya vimos lo tenemos dentro de las dependencias del fichero package.json y nos permitirá crear interfaces.
- Línea 2: importa el módulo de React-dom que como ya vimos lo tenemos dentro de las dependencias del fichero package.json y nos permitirá crear interfaces para el navegador/web.
- Línea 3: importamos un CSS debido a que React utiliza Web Pack para importa archivos CSS dentro de JS. También podemos importar otros archivos como fuentes, imágenes, logos, etc. El archivo index.css encuentra dentro del directorio src también.

```

# index.css  X
src > # index.css > body
1  body {
2    margin: 0;
3    font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", "Roboto", "Oxygen",
4      "Ubuntu", "Cantarell", "Fira Sans", "Droid Sans", "Helvetica Neue",
5      sans-serif;
6    -webkit-font-smoothing: antialiased;
7    -moz-osx-font-smoothing: grayscale;
8  }
9
10 code {
11   font-family: source-code-pro, Menlo, Monaco, Consolas, "Courier New",
12     monospace;
13 }

```

- Línea 5: `serviceWorker` que está relacionado con el concepto de `ProgressiveWebApp` un concepto que nos permite enviar notificaciones, ejecutar procesos en segundo plano, etc. Otorgándole un comportamiento muy similar al de las aplicaciones de escritorio/móvil.
- Línea 7: es la que hace el uso de `React` y la que le añade el código que falta al documento `HTML` (la imagen, el párrafo y el enlace). Lo que realmente hace `ReactDOM.render` (que quiero pintar, donde lo quiero pintar) es añadir un componente dentro del elemento del elemento con id `"root"` del `index.html` mediante a la instrucción de `JavaScript` `document.getElementById("root")`.
- Línea 4: tenemos el `import` a `App` que está llamando al fichero `App.js` del directorio `src`.