

Gestor de colas en parques temáticos

“Fun with Queues”



Jose Hurtado Baeza - 48824963B

Wilmer Fabricio Bravo Shuira - 29577276N

Sistemas Distribuidos

Práctica Julio: Gestor de colas en parques temáticos

Curso 2021-2022

Entorno de desarrollo	2
Informe de los componentes software desarrollados	5
parque_atracciones(Base de Datos)	5
Fwq_registry	9
Fwq_sensor	22
Fwq_waitingTimeServer	26
Fwq_visitor	33
Fwq_engine	59
Api_Engine	82
Front-End	86
Guía de despliegue de la aplicación	89
Capturas de pantalla que muestran el funcionamiento de las distintas aplicaciones conectadas	91

Entorno de desarrollo

Visual Studio Code

Visual Studio Code es un editor de código fuente desarrollado por Microsoft para Windows, Linux, macOS y web de código abierto para ofrecer a los usuarios una herramienta de programación avanzada. Incluye soporte para la depuración, control integrado de Git, resaltado de sintaxis, finalización inteligente de código, fragmentos y refactorización de código. También es personalizable, por lo que los usuarios pueden cambiar el tema del editor, los atajos de teclado y las preferencias.



Lenguaje de programación golang

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++. Este lenguaje ha sido desarrollado por Google.

Go es un lenguaje de programación compilado, concurrente, imperativo, estructurado, orientado a objetos y con recolector de basura que de momento es soportado por diferentes tipos de sistemas operativos. Las arquitecturas soportadas son i386, amd64 y ARM.



Apache Kafka

Apache Kafka es una plataforma de transmisión de eventos distribuida de código abierto utilizada por miles de empresas para canalizaciones de datos de alto rendimiento, análisis de transmisión, integración de datos y aplicaciones de misión crítica.



MySQL

Es un sistema de gestión de bases de datos relacional.



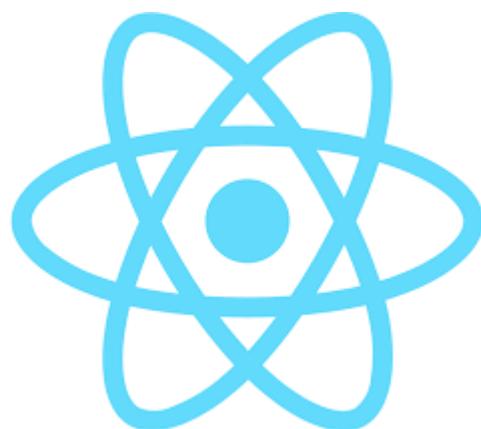
MySQL Workbench

Es una herramienta visual para desarrolladores, administradores de bases de datos, etc. MySQL Workbench proporciona modelado de datos, desarrollo SQL y herramientas de administración integrales para la configuración del servidor, administración de usuarios, respaldo y mucho más.



ReactJS

Es una biblioteca Javascript de código abierto diseñada para la creación de interfaces de usuario cuyo objetivo es facilitar el desarrollo de aplicaciones en una única página. Actualmente es mantenido por Facebook y la comunidad de software libre. A día de hoy hay más de 1000 desarrolladores libres en el proyecto.



Informe de los componentes software desarrollados

parque_atracciones(Base de Datos)

Este es el nombre de la base de datos relacional que hemos implementado en el entorno MySQL Workbench. Esta sirve para almacenar los recursos compartidos entre los distintos microservicios que forman el sistema central del parque.

Dicha base de datos fue creada mediante la interfaz de MySQL WorkBench y tras esto creamos un script llamado ParqueAtracciones.sql que es el que utilizamos para crear y llenar las tablas que nos hacían falta.

Vamos a comenzar por las sentencias SQL de creación de tablas que son las realmente importantes, ya que las inserciones se realizaron de una forma más o menos “aleatoria”.

```
/* Creación de tablas */
CREATE TABLE parque (id varchar(30) PRIMARY KEY, aforoMaximo int, aforoActual int);
```

La primera sentencia es para la creación de la sencilla tabla “parque” en la que almacenamos en la columna “id” el identificador del parque, en la columna “aforoMaximo” el máximo número de visitantes que pueden estar en el parque en un mismo momento y en la columna “aforoActual” el número de visitantes que se encuentra en el parque actualmente.

```
CREATE TABLE visitante (
    id varchar(20) PRIMARY KEY,
    nombre varchar(30) NOT NULL,
    contraseña varchar(100) NOT NULL,
    posicionx int DEFAULT 0,
    posiciony int DEFAULT 0,
    destinox int DEFAULT -1,
    destinoy int DEFAULT -1,
    dentroParque int DEFAULT 0,
    idEnParque char(1),
    ultimoEvento varchar(150),
    parqueAtracciones varchar(30) default 'SDpark',
    CONSTRAINT fk_visitantes_parque FOREIGN KEY (parqueAtracciones) REFERENCES parque (id));
```

La segunda sentencia crea la tabla “visitante” que nos va a servir para almacenar la información de los visitantes que entran al parque de atracciones.

En dicha tabla generamos la columna “id” para almacenar el identificador de cada visitante e indicamos que esta columna es clave primaria, ya que no nos interesa que haya 2 o más visitantes con el mismo identificador, sino no podríamos distinguirlos.

Después generamos las columnas “nombre” y “contraseña” para ya tener por completo las credenciales de acceso. Estas las declaramos como NOT NULL, ya que un visitante tiene que tener un nombre y un password asociado obligatoriamente para poder entrar al parque.

A continuación generamos las columnas “posicionx” y “posiciony” que nos indicarán en todo momento en qué lugar del parque se encuentra el visitante. A estas columnas les hemos añadido el valor 0 por defecto, ya que en nuestra implementación hemos decidido que la posición 0,0 sea la entrada al parque de atracciones.

Seguidamente tenemos las columnas “destinox” y “destinoy” las cuales hemos declarado con valor por defecto -1, ya que inicialmente el visitante no sabe a qué atracción va a querer dirigirse. Y por último, la columna parqueAtracciones que nos sirve para almacenar el parque de atracciones en el que se encuentra el visitante.

Luego tenemos la columna “dentroParque” que nos sirve para saber si el visitante ha iniciado sesión y se encuentra dentro del parque, para lo cual le daremos 1 como valor y 0 en caso contrario.

Acabando con las columnas de esta tabla, tenemos la columna “idEnParque” que almacena el identificador que se le asigna al visitante cuando entra en el parque y así podamos distinguirlo en el mapa a continuación y la columna “ultimoEvento” que nos sirve para la implementación del registro de auditoría solicitado en el Registry. En esta columna almacenamos los eventos de log que se producen en el visitante.

Además declaramos una clave ajena para hacer referencia al parque en el que se encuentra el visitante.

```
> CREATE TABLE atraccion(
    id varchar(30) PRIMARY KEY,
    tciclo int,
    nvisitantes int,
    posicionx int,
    posiciony int,
    tiempoEspera int,
    estado varchar(10) default 'Abierta',
    parqueAtracciones varchar(30),
    - CONSTRAINT fk_atracciones_parque FOREIGN KEY (parqueAtracciones) REFERENCES parque (id));
```

La tercera sentencia de creación de tablas es para la tabla “atraccion” que nos va a servir para almacenar la información de las atracciones que vamos a tener en nuestro parque.

Entrando en detalle, tenemos una columna “id” para guardar el identificador de la atracción, esta columna va a ser la clave primaria de la tabla, ya que no tiene sentido que 2 o más atracciones tengan el mismo identificador, ya que no podríamos distinguirlas fácilmente.

A continuación, tenemos la columna “tciclo” en la que almacenamos el tiempo (en minutos) que tarda la atracción en completar un ciclo de su funcionamiento y la columna “nvisitantes”

que nos indica cuántas personas o más bien visitantes pueden montarse en la atracción en cada ciclo.

Seguidamente tenemos las columnas “posicionx” y “posiciony” que nos permiten saber en qué lugar del parque se encuentra instalada la atracción. Luego la columna “tiempoEspera” en la que almacenaremos el tiempo de espera de la atracción calculado por el Engine en base a la información proporcionada por el servidor de tiempos de espera, después tenemos la columna “estado” que va a ir almacenando si la atracción está abierta o cerrada dependiendo del clima de la ciudad en la que se encuentra su cuadrante y finalmente la columna “parqueAtracciones” que declaramos como clave ajena ya que hace referencia al parque en el que se encuentra la atracción.

La cuarta sentencia de creación de tabla es para la tabla mapa:

```
CREATE TABLE mapa(fila int(2) PRIMARY KEY, infoParque varchar(100));
```

En dicha tabla iremos almacenando el estado actual del mapa del parque, y tenemos dos columnas: “fila” que es un simple identificador numérico para localizar los elementos del mapa desde el front y infoParque donde introduciremos el contenido de las 20 filas que forman el mapa del parque.

Y la quinta y última sentencia de creación de tabla es para la tabla ciudades:

```
CREATE TABLE ciudades(cuadrante varchar(30) PRIMARY KEY, nombre varchar(30), temperatura float);
```

Con esta tabla lo que pretendemos es almacenar la información básica de las ciudades para poder consultarla en tiempo real desde el front. En esta tabla tenemos una columna que actúa como clave principal llamada “cuadrante” en la que indicaremos a qué cuadrante del parque corresponde la ciudad, si “arriba-izquierda”, “abajo-derecha”, “arriba-derecha” o “abajo-izquierda” respectivamente, por otro lado, tenemos la columna “nombre” en la que almacenamos el nombre de la ciudad y la columna “temperatura” donde almacenamos la temperatura de la ciudad.

Por otro lado, tenemos los insert para la tabla atracción y es que en nuestro caso tenemos 16 atracciones:

```
/* Inserciones en las tablas */
INSERT INTO parque (id, aforoActual) VALUES ('SDpark', 0);
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion1", 5, 3, 10, 14, 45, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion2", 8, 9, 1, 4, 30, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion3", 7, 6, 6, 6, 15, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion4", 18, 9, 10, 19, 65, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion5", 4, 5, 9, 17, 10, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion6", 10, 6, 3, 18, 40, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion7", 11, 8, 9, 2, 80, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion8", 19, 7, 2, 3, 90, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion9", 14, 6, 7, 8, 20, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion10", 15, 4, 18, 11, 13, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion11", 17, 7, 17, 5, 7, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion12", 7, 8, 6, 5, 17, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion13", 8, 2, 16, 17, 77, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion14", 12, 4, 19, 18, 40, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion15", 13, 10, 4, 15, 74, "SDpark");
INSERT INTO atraccion (id, tciclo, nvisitantes, posicionx, posiciony, tiempoEspera, parqueAtracciones) VALUES ("atraccion16", 19, 11, 15, 15, 23, "SDpark");
```

Y para terminar con el script sql, tenemos los insert de la tabla mapa con la situación inicial del parque. Esta tabla la iremos actualizando para que desde el front podamos seguir el estado en tiempo real del mapa del parque.

```
INSERT INTO mapa (fila, infoParque) VALUES (0, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (1, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (2, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (3, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (4, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (5, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (6, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (7, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (8, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (9, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (10, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (11, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (12, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (13, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (14, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (15, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (16, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (17, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (18, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
INSERT INTO mapa (fila, infoParque) VALUES (19, "-----|-----|-----|-----|-----|-----|-----|-----|-----|-----");
```

Fwq_registry

El primer microservicio que vamos a comentar es el Registry, este se encargará del registro de los visitantes en el parque, tanto por sockets como por API REST.

```
You, 2 weeks ago | 1 author (You)
package main

import (
    "bufio"
    "crypto/rand"
    "crypto/tls"
    "database/sql"
    "encoding/json"
    "fmt"
    "log"
    "net"
    "net/http"
    "os"
    "strconv"
    "strings"
    "time"

    _ "github.com/go-sql-driver/mysql"
    "github.com/gorilla/mux"
    "github.com/kabukky/httpscerts"
    "golang.org/x/crypto/bcrypt"
)

/*
 * Estructura del visitante
 */
You, 2 weeks ago | 1 author (You)
type visitante struct {
    ID          string `json:"id"`
    Nombre      string `json:"nombre"`
    Password    string `json:"contraseña"`
    Posicionx   int    `json:"posicionx"`
    Posiciony   int    `json:"posiciony"`
    Destinox    int    `json:"destinox"`
    Destinoy    int    `json:"destinoy"`
    DentroParque int   `json:"dentroParque"`
    IdEnParque  string `json:"idEnParque"`
    UltimoEvento string `json:"ultimoEvento"`
    Parque      string `json:"parqueAtracciones"`
}
```

En esta primera captura del código hemos podido ver la declaración del paquete, la importación de librerías que hemos necesitado y la creación del struct del visitante.

```

func main() {
    host := os.Args[1]
    puertoSockets := os.Args[2]
    puertoAPI := os.Args[3]

    cert, err := tls.LoadX509KeyPair("cert/cert.pem", "cert/key.pem")
    if err != nil {
        log.Fatal(err)
    }

    config := tls.Config{
        Certificates: []tls.Certificate{cert},
        ClientAuth:   tls.RequireAnyClientCert,
    }

    config.Rand = rand.Reader

    // Arrancamos el servidor y atendemos conexiones entrantes
    fmt.Println("Arrancando el Registry, atendiendo vía sockets en " + host + ":" + puertoSockets)

    //l, err := net.Listen("tcp", host+":"+puerto) // CONEXIONES INSEGURAS
    l, err := tls.Listen("tcp", host+":"+puertoSockets, &config) // CONEXIONES SEGURAS
    if err != nil {
        log.Fatal("Error escuchando", err.Error())
    }

    // Cerramos el listener cuando se cierra la aplicación
    defer l.Close()

    go lanzarServidor(host, puertoAPI) // El servidor funcionará de forma paralela y concurrente a los sockets

    // Bucle infinito hasta la salida del programa
    for {

        // Atendemos conexiones entrantes
        conn, err := l.Accept()
        if err != nil {
            log.Println("Error conectando con un visitante: ", err.Error())
            continue
        }

        // Imprimimos la dirección de conexión del cliente
        log.Println("Visitante " + conn.RemoteAddr().String() + " conectado.")

        // Llamamos a la función de forma asíncrona y manejamos las conexiones de forma concurrente
        go manejoConexion(conn)
    }
}

```

En esta segunda captura tenemos el bloque de código del main, donde primero guardamos la información pasada por parámetro, concretamente la ip, el puerto de escucha vía sockets y el puerto de escucha del api rest.

Seguidamente cargamos el certificado autofirmado en la configuración de tls para poder implementar una comunicación segura en los sockets.

A continuación, tenemos la parte donde arrancamos el registrador para que escuche por la ip y puerto indicados por parámetro, controlando el error en caso de fallo al ejecutar el tls.Listen. Y además asegurándonos de que el listener se cerrará cuando se acabe el programa, esto lo hacemos con la declaración defer.

Después de esto, nos encontramos con un bucle infinito, ya que nos interesa que el Registry esté activo siempre para recibir peticiones por parte de los visitantes, para lo que

tenemos la función Accept(). Al igual que antes, controlamos en caso de error imprimiendo un mensaje por pantalla relacionado y en caso de que no se produzca dicho error, imprimimos por pantalla la dirección de conexión del visitante que se ha conectado.

Finalmente llamamos de forma asíncrona a la función manejoConexion que trabajará de forma concurrente para poder atender a todos los visitantes que se conecten al Registry vía sockets. En la llamada pasamos la conexión establecida vía socket.

```
/* Función que procesa concurrentemente los registros o actualizaciones de los visitantes */
func manejoConexion(conexion net.Conn) {
    // Lectura de la opción elegida por el visitante
    opcion, err := bufio.NewReader(conexion).ReadBytes('\n')

    // Cerramos la conexión de los clientes que se han desconectado
    if err != nil {
        fmt.Println("Visitante " + conexion.RemoteAddr().String() + " desconectado.")
        conexion.Close()
        return
    }
    You, a month ago • Creación y edición de perfiles ajustada, ahora el...
    opcionElegida := strings.TrimSpace(string(opcion))

    // Lectura del id del visitante hasta final de línea
    id, err := bufio.NewReader(conexion).ReadBytes('\n')

    // Cerramos la conexión de los clientes que se han desconectado
    if err != nil {
        fmt.Println("Visitante " + conexion.RemoteAddr().String() + " desconectado.")
        conexion.Close()
        return
    }

    // Lectura del nombre del visitante hasta final de línea
    nombre, err := bufio.NewReader(conexion).ReadBytes('\n')

    // Cerramos la conexión de los clientes que se han desconectado
    if err != nil {
        fmt.Println("Visitante " + conexion.RemoteAddr().String() + " desconectado.")
        conexion.Close()
        return
    }

    // Lectura del password del visitante hasta final de línea
    password, err := bufio.NewReader(conexion).ReadBytes('\n')

    // Cerramos la conexión de los clientes que se han desconectado
    if err != nil {
        fmt.Println("Visitante " + conexion.RemoteAddr().String() + " desconectado.")
        conexion.Close()
        return
    }
```

Entrando en el código de la función, aquí hacemos 4 lecturas a través de la conexión vía socket establecida con el Visitante, leyendo la opción elegida, el id, el nombre y el password del visitante que va a ser registrado o va a ser modificado, controlando en cada una de las lecturas si el visitante se ha desconectado para mostrar el mensaje de alerta.

```

// Si se ha solicitado un registro de usuario
if opcionElegida == "1" {

    // Imprimimos la información del visitante a registrar
    fmt.Println("Visitante a registrar -> ID: " + strings.TrimSpace(string(id)) +
    " | Nombre: " + strings.TrimSpace(string(nombre)) + " | Password: " + strings.TrimSpace(string(password)))

    // Si se ha solicitado editar/actualizar un perfil de usuario existente
} else if opcionElegida == "2" {

    // Imprimimos la información del visitante a editar
    fmt.Println("Visitante a editar -> ID: " + strings.TrimSpace(string(id)) +
    " | Nombre: " + strings.TrimSpace(string(nombre)) + " | Password: " + strings.TrimSpace(string(password)))

} else {
    conexion.Write([]byte("La opción elegida no es válida"))
    conexion.Close()
}

// Accedemos a la base de datos, empezando por abrir la conexión
db, err := sql.Open("mysql", "root:1234@tcp(127.0.0.1:3306)/parque_atracciones")

// Comprobamos que no haya error al conectarse
if err != nil {
    panic("Error al conectarse con la BD: " + err.Error())
}

defer db.Close() // Para que siempre se cierre la conexión con la BD al finalizar el programa

v := visitante{
    ID:      strings.TrimSpace(string(id)),
    Nombre:  strings.TrimSpace(string(nombre)),
    Password: strings.TrimSpace(string(password)),
    IdParque: strings.TrimSpace(string(id[0])),
}

```

Siguiendo con el código de la función, dependiendo de si se ha solicitado un registro o una modificación sacamos por pantalla la información del visitante de forma personalizada, y en caso contrario mostramos un error y cerramos la conexión. Si todo va bien abrimos la conexión a la base de datos, controlando con un panic para cerrar el programa en caso de que no se pueda acceder a la BD, añadiendo a continuación un defer para asegurarnos de que cuando finalice el programa se cierre siempre dicha conexión a la base de datos. A continuación creamos una instancia del struct visitante que nos va a servir para tratar la información del visitante.

```

// Si se ha solicitado un registro
if opcionElegida == "1" {

    results, err := db.Query("SELECT * FROM visitante WHERE id = ?", v.ID) // Comprueba si el visitante está registrado en la aplicación

    // Comprobamos que no se produzcan errores al hacer la consulta
    if err != nil {
        panic("Error al hacer la consulta de la información del visitante indicado: " + err.Error())
    }

    defer results.Close() // Nos aseguramos de cerrar

    // Si el visitante ya se había registrado
    if results.Next() {

        conexion.Write([]byte("ERROR: El visitante ya estaba registrado en la aplicación"))
        conexion.Close()

        RegistroLog(db, conexion.RemoteAddr().String(), v.ID, "Error", "El visitante "+v.ID+" ya estaba registrado en la aplicación") // Registraremos el evento de log

    } else { // Si es un nuevo usuario

        results, err := db.Query("SELECT * FROM visitante") // Devuelve los visitantes que hay registrados en la aplicación

        // Comprobamos que no se produzcan errores al hacer la consulta
        if err != nil {
            panic("Error al hacer la consulta a la BD: " + err.Error())
        }

        defer results.Close() // Nos aseguramos de cerrar

        // Nos guardamos el nº actual de visitantes registrados en la aplicación
        visitantesActuales := 0
        for results.Next() {
            visitantesActuales += 1
        }

        // INSERTAMOS el nuevo visitante en la BD
        // Preparamos para prevenir inyecciones SQL
        sentenciaPreparada, err := db.Prepare("INSERT INTO visitante (id, nombre, contraseña, idEnParque) VALUES (?, ?, ?, ?)")
        if err != nil {
            panic("Error al preparar la sentencia de inserción: " + err.Error())
        }

        defer sentenciaPreparada.Close()

        // Ejecutar sentencia, un valor por cada '?'
        _, err = sentenciaPreparada.Exec(v.ID, v.Nombre, HashPassword(v.Password), v.IdEnParque)
        if err != nil {
            panic("Error al registrar el visitante: " + err.Error())
        }

        conexion.Write([]byte("Visitante registrado en el parque."))
        fmt.Println("Registro completado. Actualmente hay " + strconv.Itoa(visitantesActuales+1) + " visitantes registrados.")
        conexion.Close()

        RegistroLog(db, conexion.RemoteAddr().String(), v.ID, "Alta", "Visitante "+v.ID+" registrado correctamente") // Registraremos el evento de log
    }
}

```

Entonces, si se ha solicitado un registro, lo primero que hacemos es una consulta a la base de datos pidiendo que nos devuelva las filas de la tabla visitante donde el id se corresponda con el recibido por el visitante vía socket. Controlamos el error al hacer la consulta y nos aseguramos de cerrar la consulta a la base de datos con otro defer. Y gracias a esta consulta podemos comprobar si el visitante ya se encuentra registrado, en cuyo caso cerramos la conexión, no sin antes devolver un error a la aplicación visitante indicando el problema mencionado y registrando el evento de log correspondiente en la tabla de dicho visitante.

Por el contrario, si no se encuentra registrado, entonces procederemos a calcular el número de visitantes registrados para después insertar en la base de datos al nuevo visitante y así enviar un mensaje a la aplicación visitante en la que se informe que el visitante se ha registrado correctamente y además indicando el número total de visitantes registrados actualmente, finalizando con el cierre de la conexión y el posterior registro de log.

```

    }

} else { // Si se ha solicitado una actualización
    You, 7 months ago • Actualizado el registry y el visitante para que a...
    results, err := db.Query("SELECT * FROM visitante WHERE id = ?", v.ID) // Comprueba si el visitante está registrado en la aplicación

    // Comprobamos que no se produzcan errores al hacer la consulta
    if err != nil {
        panic("Error al hacer la consulta de la información del visitante indicado: " + err.Error())
    }

    defer results.Close() // Nos aseguramos de cerrar

    // Si el ID del visitante indicado por el cliente existe
    if results.Next() {

        // MODIFICAMOS la información de dicho visitante en la BD
        // Preparamos para prevenir inyecciones SQL
        sentenciaPreparada, err := db.Prepare("UPDATE visitante SET nombre = ?, contraseña = ? WHERE id = ?")
        if err != nil {
            panic("Error al preparar la sentencia de actualización: " + err.Error())
        }

        defer sentenciaPreparada.Close()

        // Ejecutar sentencia, un valor por cada '?'
        _, err = sentenciaPreparada.Exec(v.Nombre, HashPassword(v.Password), v.ID)
        if err != nil {
            panic("Error al modificar el visitante: " + err.Error())
        }

        conexion.Write([]byte("Visitante actualizado correctamente"))
        conexion.Close()

        RegistroLog(db, conexion.RemoteAddr().String(), v.ID, "Modificación", "Visitante "+v.ID+" actualizado correctamente") // Registramos el evento de log

    } else {
        conexion.Write([]byte("El id del visitante no existe"))
        conexion.Close()
    }
}

// Reiniciamos el proceso
manejoConexion(conexion)
}

```

Por otra parte, si la petición es una modificación de la información de un visitante empezamos comprobando que el visitante se haya registrado previamente, después si es así, preparamos la sentencia a ejecutar, de tal manera que prevenimos inyecciones SQL, controlando en caso de error que se finalice el programa y se cierre la nueva conexión a la base de datos, y en caso de que todo haya ido bien, se ejecutará la sentencia que hemos preparado sustituyendo las interrogaciones con los valores indicados en la llamada a Exec, donde al igual que antes controlamos que no se produzca un error, y si todo se ejecuta correctamente envíamos vía socket un mensaje de respuesta al visitante informando de que su información ha sido actualizada, para finalmente cerrar la conexión y registrar el evento de log. Si por el contrario, el visitante indicado en la petición no está registrado, enviamos un mensaje informando del error y cerramos la conexión sin hacer nada más.

Lo último que hacemos en el código de la función manejoConexion es precisamente hacer una llamada recursiva, de tal forma que el Registry se mantendrá en todo momento a la espera de peticiones por parte del visitante y manejando las conexiones con los visitantes de forma concurrente.

Cabe mencionar además, que las contraseñas las estamos almacenando en la BD utilizando esta función de hash:

```

/* Función que genera y devuelve el hash de la contraseña pasada por parámetro */
func HashPassword(password string) string {

    hash, err := bcrypt.GenerateFromPassword([]byte(password), bcrypt.DefaultCost)
    if err != nil {
        log.Fatal(err)
    }

    return string(hash)
}

```

Así nos aseguramos la protección de dichas contraseñas en caso de robo de información de la BD, ya que el hash es un tipo de cifrado irreversible.

Por otro lado, para registrar los logs nos estamos sirviendo de esta función:

```

/* Función que almacena los registros de auditoría en la tabla visitante */
func RegistroLog(db *sql.DB, ipPuerto, idVisitante, accion, descripcion string) {

    // Añadimos el evento de log de error al visitante
    sentenciaPreparada, err := db.Prepare("UPDATE visitante SET ultimoEvento = ? WHERE id = ?")
    if err != nil {
        panic("Error al preparar la sentencia de inserción: " + err.Error())
    }

    defer sentenciaPreparada.Close()

    var eventoLog string // Variable donde vamos a guardar la información de log que le vamos a pasar a la BD

    dateTime := time.Now().Format("2006-01-02 15:04:05") // Fecha y hora del evento con formato personalizado
    //ipVisitante := conexion.RemoteAddr().String()
    ipVisitante := ipPuerto // IP y puerto de quién ha provocado el evento
    accionRealizada := accion // Que acción se realiza
    descripcionEvento := descripcion // Parámetros o descripción del evento

    eventoLog += dateTime + " | "
    eventoLog += ipVisitante + " | "
    eventoLog += accionRealizada + " | "
    eventoLog += descripcionEvento

    _, err = sentenciaPreparada.Exec(eventoLog, idVisitante)
    if err != nil {
        panic("Error al registrar el evento de log: " + err.Error())
    }
}

```

La cual, almacena en la columna “ultimoEvento” de la tabla Visitante la información del último suceso asociado a cada visitante en concreto.

Con esto ya tendríamos explicado el bloque correspondiente a las peticiones vía socket, ahora vamos con las peticiones vía API REST.

Si nos fijamos en el main, tenemos una función llamada “lanzarServidor” y esta función concretamente lanza un servidor API REST que funcionará de forma paralela y concurrente a los sockets.

```

/* Función que se encarga de arrancar el servidor API REST */
func lanzarServidor(host, puertoAPI string) {

    // Comprobamos si los ficheros de certificado están disponibles
    err := httpscerts.Check("cert.pem", "key.pem")

    // Si no están disponibles, generamos unos nuevos
    if err != nil {
        err = httpscerts.Generate("cert.pem", "key.pem", host+":"+puertoAPI)
        if err != nil {
            log.Fatal("ERROR: No se pudieron crear los certificados https.")
        }
    }

    // IMPLEMENTAMOS EL API REST
    // Rutas
    mux := mux.NewRouter()

    // Responder al cliente
    mux.HandleFunc("/crear/{id:[A-Za-z0-9_]+}", crearPerfil).Methods("POST")
    mux.HandleFunc("/editar/{id:[A-Za-z0-9_]+}", editarPerfil).Methods("PUT")

    // SERVIDOR
    // Arrancamos el servidor https en una go routine
    go http.ListenAndServeTLS(host+":"+puertoAPI, "cert.pem", "key.pem", mux)
    fmt.Println("Servidor API REST corriendo en https://" + host + ":" + puertoAPI)
    //log.Fatal(http.ListenAndServe(": "+puertoAPI, http.HandlerFunc(redirectToHttps)))
}

}

```

En esta función empezamos comprobando si existe un certificado autofirmado para asegurar la conexión vía https y sino lo está lo generamos.

Tras esto empezamos la implementación del api rest sirviéndose de una librería llamada Gorilla Mux. Entonces definimos las posibles URLs entrantes y sus correspondientes manejadores y arrancamos el servidor https en una go routine para no estorbar a los sockets. Ya que esto garantiza la concurrencia.

Antes de entrar en los manejadores, vamos a comentar una serie de funciones auxiliares que hemos utilizado para formar las respuestas http a las peticiones a nuestro servidor API REST. Para ello, en primer lugar nos hemos definido una estructura con un formato más o menos estándar del contenido de una respuesta http:

```

// INICIO BLOQUE RESPONSE

// Estructura con el formato de una respuesta http
You, last week | 1 author (You)
type Response struct {
    Status      int         `json:"status"`
    Data        interface{} `json:"data"`
    Message     string      `json:"message"`
    contentType string
    responseWrite http.ResponseWriter
}

```

En segundo lugar tenemos estas dos funciones:

```
/* Función que crea una respuesta por defecto para los clientes de la API REST */
func CreateDefaultResponse(rw http.ResponseWriter) Response {
    return Response{
        Status:         http.StatusOK,
        responseWrite: rw,
        contentType:   "application/json",
    }
}

/* Función que envía las respuestas a los clientes de la API REST */
func (resp *Response) Send() {
    resp.responseWrite.Header().Set("Content-Type", resp.contentType)
    resp.responseWrite.WriteHeader(resp.Status)

    // Marshall devuelve 2 valores: Los valores transformados en tipo byte y un error
    output, _ := json.Marshal(&resp) // Para responder con json
    //output, _ := xml.Marshal(&resp) // Para responder con xml
    //output, _ := yaml.Marshal(&resp) // Para responder con yaml
    fmt.Fprintln(resp.responseWrite, string(output))
}
```

La primera nos crea una respuesta por defecto para los clientes y la segunda se encarga de realizar el envío de las respuestas a dichos clientes. En nuestro caso, respondemos en formato json.

```
/* Función que envía una respuesta a los clientes indicando que el registro ha sido satisfactorio */
func SendDataCrearPerfil(rw http.ResponseWriter, data interface{}) {
    response := CreateDefaultResponse(rw)
    //response.Data = data
    response.Message = "OK: Visitante registrado correctamente"
    response.Send()
}

/* Función que envía una respuesta a los clientes indicando que el registro ha sido satisfactorio */
func SendDataEditarPerfil(rw http.ResponseWriter, data interface{}) {
    response := CreateDefaultResponse(rw)
    //response.Data = data
    response.Message = "OK: Visitante modificado correctamente"
    response.Send()
}
```

A continuación tenemos estas dos, las cuales se sirven de las 2 anteriores para enviar respuestas personalizadas dependiendo de si se ha recibido una petición de creación de perfil o de modificación. En ambos casos, estas las utilizaremos cuándo la operación se haya resuelto de forma satisfactoria y sin inconvenientes.

```

/* Función utilizada junto a la de abajo cuando no se encuentra el recurso solicitado. */
func (resp *Response) NotFound() {
    resp.Status = http.StatusNotFound
    resp.Message = "ERROR: Resource Not Found"
}

/* Función que manda una respuesta indicando que el recurso solicitado no ha sido encontrado */
func SendNotFound(rw http.ResponseWriter) {
    response := CreateDefaultResponse(rw)
    response.NotFound()
    response.Send()
}

```

Luego tenemos estas dos que podemos usar cuando el recurso solicitado no se ha encontrado.

```

/* Función que prepara la respuesta para cuando el id ya existe y se pide un registro sobre dicho id */
func (resp *Response) YaExiste() {
    resp.Status = http.StatusBadRequest
    resp.Message = "ERROR: El visitante ya estaba registrado"
}

/* Función que manda una respuesta indicando que el id indicado ya existe */
func SendYaExiste(rw http.ResponseWriter) {
    response := CreateDefaultResponse(rw)
    response.YaExiste()
    response.Send()
}

```

Estas dos para responder cuándo se solicite un registro y ya exista el id indicado en la URI.

```

/* Función que prepara la respuesta para cuando el id no existe y se pide una modificación sobre dicho id */
func (resp *Response) NoExiste() {
    resp.Status = http.StatusBadRequest
    resp.Message = "ERROR: El id del visitante indicado no corresponde con uno existente"
}

/* Función que manda una respuesta indicando que el id indicado no existe */
func SendNoExiste(rw http.ResponseWriter) {
    response := CreateDefaultResponse(rw)
    response.NoExiste()
    response.Send()
}

```

Estas dos para cuando se solicita una modificación de perfil sobre un id no registrado.

```

/* Función utilizada junto a la de abajo al momento de insertar o
actualizar una fila de la BD y que se produzcan errores para poder manejarlos. */
func (resp *Response) UnprocessableEntity() {
    resp.Status = http.StatusUnprocessableEntity
    resp.Message = "ERROR: UnprocessableEntity Not Found"
}

/* Función que manda una respuesta indicando que la entidad recibida no es procesable */
func SendUnprocessableEntity(rw http.ResponseWriter) {
    response := CreateDefaultResponse(rw)
    response.UnprocessableEntity()
    response.Send()
}

```

Y estas dos para cuando se ha recibido un body no válido para el registro o la modificación de un usuario. Ya que en dicho body deben venir el nombre y la contraseña del usuario en un formato adecuado.

Y por último nos queda hablar de los 2 manejadores. En ambos casos la lógica utilizada es similar a la de los sockets por lo que no vamos a entrar mucho en detalle.

```

/* Función manejadora para la creación del perfil de un visitante */
func crearPerfil(rw http.ResponseWriter, r *http.Request) {
    log.Println("Petición de creación de perfil recibida -> " + r.URL.Path)

    //Obtener ID
    vars := mux.Vars(r)
    userId := vars["id"]

    v := visitante{}
    v.ID = userId
    decoder := json.NewDecoder(r.Body)

    if err := decoder.Decode(&v); err != nil {
        SendUnprocessableEntity(rw)
    } else {
        // Insertamos el nuevo visitante en la BD

        // Accedemos a la base de datos, empezando por abrir la conexión
        db, err := sql.Open("mysql", "root:1234@tcp(127.0.0.1:3306)/parque_atracciones")

        // Comprobamos que no haya error al conectarse
        if err != nil {
            panic("Error al conectarse con la BD: " + err.Error())
        }

        defer db.Close() // Para que siempre se cierre la conexión con la BD al finalizar el programa

        results, err := db.Query("SELECT * FROM visitante WHERE id = ?", v.ID) // Comprueba si el visitante está registrado en la aplicación
        You, last week * Registry: Manejador para crear perfil casi termin...
        // Comprobamos que no se produzcan errores al hacer la consulta
        if err != nil {
            panic("Error al hacer la consulta de la información del visitante indicado: " + err.Error())
        }

        defer results.Close() // Nos aseguramos de cerrar
    }
}

```

En este primero destacamos la obtención del id usando mux y la decodificación del body para obtener el nombre y la contraseña del usuario a registrar. Informando al respecto al cliente en caso de que la entidad recibida no sea procesable.

```

// Si el visitante ya se había registrado
if results.Next() {
    SendYaExiste(rw)

    RegistroLog(db, r.RemoteAddr, v.ID, "Error", "El visitante "+v.ID+" ya estaba registrado en la aplicación") // Registraremos el evento de log
} else { // Si es un nuevo usuario
    results, err := db.Query("SELECT * FROM visitante") // Devuelve los visitantes que hay registrados en la aplicación

    // Comprobamos que no se produzcan errores al hacer la consulta
    if err != nil {
        panic("Error al hacer la consulta a la BD: " + err.Error())
    }

    defer results.Close() // Nos aseguramos de cerrar

    // Nos guardamos el nº actual de visitantes registrados en la aplicación
    visitantesActuales := 0
    for results.Next() {
        visitantesActuales += 1
    }

    // INSERTAMOS el nuevo visitante en la BD
    // Preparamos para prevenir inyecciones SQL
    sentenciaPreparada, err := db.Prepare("INSERT INTO visitante (id, nombre, contraseña, idEnParque) VALUES(?, ?, ?, ?)")
    if err != nil {
        panic("Error al preparar la sentencia de inserción: " + err.Error())
    }

    defer sentenciaPreparada.Close()

    fmt.Println("Visitante a registrar -> ID: ", v.ID, "Nombre: ", v.Nombre, "Password: ", v.Password)

    // Ejecutar sentencia, un valor por cada '?'
    _, err = sentenciaPreparada.Exec(v.ID, v.Nombre, HashPassword(v.Password), string(v.ID[0]))
    if err != nil {
        panic("Error al registrar el visitante: " + err.Error())
    }

    fmt.Println("Registro completado. Actualmente hay " + strconv.Itoa(visitantesActuales+1) + " visitantes registrados.")

    RegistroLog(db, r.RemoteAddr, v.ID, "Alta", "Visitante "+v.ID+" registrado correctamente") // Registraremos el evento de log
    SendDataCrearPerfil(rw, v)
}

}

```

Posteriormente en caso de que el visitante estuviera registrado informamos de esta situación al cliente con la función antes descrita y si no procedemos de forma similar a los sockets salvo que al final enviamos una respuesta informando al cliente de que el registro se ha realizado de forma satisfactoria.

```

/* Función manejadora para la modificación del perfil de un visitante */
func editarPerfil(rw http.ResponseWriter, r *http.Request) {
    log.Println("Petición de modificación de perfil recibida -> " + r.URL.Path)

    // Accedemos a la base de datos, empezando por abrir la conexión
    db, err := sql.Open("mysql", "root:1234@tcp(127.0.0.1:3306)/parque_atracciones")

    // Comprobamos que no haya error al conectarse
    if err != nil {
        panic("Error al conectarse con la BD: " + err.Error())
    }

    defer db.Close() // Para que siempre se cierre la conexión con la BD al finalizar el programa

    //Obtener ID
    vars := mux.Vars(r)
    userId := vars["id"]
    v := visitante{}
    v.ID = userId

    results, err := db.Query("SELECT * FROM visitante WHERE id = ?", v.ID) // Comprueba si el visitante está registrado en la aplicación

    // Comprobamos que no se produzcan errores al hacer la consulta
    if err != nil {
        panic("Error al hacer la consulta de la información del visitante indicado: " + err.Error())
    }

    defer results.Close() // Nos aseguramos de cerrar

    // Si el ID del visitante indicado por el cliente existe
    if results.Next() {
        decoder := json.NewDecoder(r.Body)

        if err := decoder.Decode(&v); err != nil {
            SendUnprocessableEntity(rw)
        } else {

            // MODIFICAMOS la información de dicho visitante en la BD
            // Preparamos para prevenir inyecciones SQL
            sentenciaPreparada, err := db.Prepare("UPDATE visitante SET nombre = ?, contraseña = ? WHERE id = ?")
            if err != nil {
                panic("Error al preparar la sentencia de actualización: " + err.Error())
            }

            defer sentenciaPreparada.Close()

            fmt.Println("Visitante a editar -> ID: ", v.ID, "Nombre: ", v.Nombre, "Password: ", v.Password)

            // Ejecutar sentencia, un valor por cada '?'
            _, err = sentenciaPreparada.Exec(v.Nombre, HashPassword(v.Password), v.ID)
            if err != nil {
                panic("Error al modificar el visitante: " + err.Error())
            }

            RegistroLog(db, r.RemoteAddr, v.ID, "Modificación", "Visitante "+v.ID+" actualizado correctamente") // Registraremos el evento de log
            SendDataEditarPerfil(rw, v)
        }
    } else {
        SendNoExiste(rw)
    }
}

```

En cuanto al manejador de edición de perfil, la lógica utilizada es muy similar, lo único a destacar es que enviamos los mensajes personalizados correspondientes.

Fwq_sensor

Vamos ahora con los sensores, estos son los dispositivos que se encargarán de enviar cada cierto tiempo aleatorio, el número de personas que hay en la cola de la atracción en la que dicho sensor está ubicado al servidor de tiempos de espera.

```
package main

import (
    "context"
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"

    "github.com/segmentio/kafka-go"
)

You, 5 days ago | 1 author (You)
type sensor struct {
    IdAtraccion string
    Personas    int
}
```

Empezando a comentar el código, aquí podemos ver la declaración del paquete, las importaciones de librerías necesarias y la creación del struct sensor para facilitar el almacenamiento y el manejo de la información de un sensor.

```

func main() {
    ipBrokerGestorColas := os.Args[1]
    puertoBrokerGestorColas := os.Args[2]
    idAtraccion := os.Args[3] // Convertimos a entero
    crearTopic(ipBrokerGestorColas, puertoBrokerGestorColas)

    // Comprobamos que el id de la atracción sea válido
    valido := false
    for i := 1; !valido && i < 17; i++ {
        if idAtraccion == "atraccion"+strconv.Itoa(i) {
            valido = true
        }
    }

    // Si el id pasado por parámetro no es válido
    if !valido {
        panic("Error: El id de la atracción no es válido. Introduzca atraccion1...16")
    }

    brokerAddress := ipBrokerGestorColas + ":" + puertoBrokerGestorColas

    // Creamos un sensor
    s := new(sensor)
    s.IdAtraccion = idAtraccion

    // Generamos un número aleatorio de personas inicialmente en la cola de la atracción
    rand.Seed(time.Now().UnixNano()) // Utilizamos la función Seed(semilla) para inicializar la fuente predeterminada al requerir un comportamiento determinista
    min := 0
    max := 60
    s.Personas = (rand.Intn(max-min+1) + min)
    fmt.Println("Sensor creado para la atracción (" + idAtraccion + ") en la que inicialmente hay " + strconv.Itoa(s.Personas) + " personas en cola")

    // Generamos un tiempo aleatorio entre 1 y 3 segundos
    rand.Seed(time.Now().UnixNano()) // Utilizamos la función Seed(semilla) para inicializar la fuente predeterminada al requerir un comportamiento determinista
    min = 1
    max = 3
    tiempoAleatorio := (rand.Intn(max-min+1) + min)

    // Enviamos al servidor de tiempos el número de personas que se encuentra en la cola de la atracción
    enviaInformacion(s, brokerAddress, tiempoAleatorio)

    defer func() {
        fmt.Println("El sensor ha sido apagado.")
    }()
}

```

Respecto al main, en este primero nos guardamos los valores de los 3 parámetros introducidos al ejecutar el microservicio que son: La ip del broker gestor de colas, el puerto del broker gestor de colas y el identificador de la atracción en la base de datos.

A continuación, creamos el topic con una función auxiliar que aquí no vamos a comentar, a la cual le pasamos la ip y el puerto del broker gestor de colas, y dicha función nos va a generar de forma automática el topic llamado “sensor-servidorTiempos”.

Luego comprobamos que dicho id de la atracción introducido por parámetro sea válido, ya que si no se corresponde se generará un panic que mostrará la información del error adecuadamente por pantalla.

Después formamos la brokerAddress que le pasaremos a la función “envialinformacion” para poder conectarnos al gestor de colas que es el que mandará la información de los sensores al servidor de tiempos de espera.

Creamos a continuación un “objeto” sensor al que le asignamos la atracción pasada por parámetro. Seguidamente vamos a generar un par de números aleatorios, el primero para el

número de personas en la cola de la atracción y el segundo para el tiempo aleatorio que pasa entre cada envío de información por parte del sensor.

Y finalmente llamamos a la función “enviaInformacion” pasando el objeto sensor, la broker address y el tiempo aleatorio que pasa entre cada envío de información al servidor de tiempos de espera. Además con una función asíncrona notificamos al servidor de tiempos la caída del sensor.

```
/* Función que envía mediante un productor de Kafka la información recogida por el sensor */
func enviaInformacion(s *sensor, brokerAddress string, tiempoAleatorio int) {
    // Inicializamos el escritor
    escritor := kafka.NewWriter(kafka.WriterConfig{
        Brokers: []string{brokerAddress},
        Topic:   "sensor-servidorTiempos",
    })
    You, 2 months ago * Modificada la función enviaInformacion que ya no ...
    c := make(chan os.Signal, 1)
    signal.Notify(c, os.Interrupt)
    go func() {
        for sig := range c {
            log.Printf("captured %v, stopping profiler and exiting..", sig)
            mensaje := "\nSensor de la atracción (" + s.IdAtraccion + ") desconectado\n"
            err := escritor.WriteMessages(context.Background(),
                kafka.Message{
                    Key:  []byte("Atraccion " + s.IdAtraccion),
                    Value: []byte(mensaje),
                })
            if err != nil {
                panic("Error al conectarse al gestor de colas - No se puede mandar la información al servidor de tiempos de espera: " + err.Error())
            }
            fmt.Println()
            fmt.Println("Sensor desconectado manualmente")
            pprof.StopCPUProfile()
            os.Exit(1)
        }
    }()
}

for {
    err := escritor.WriteMessages(context.Background(),
        kafka.Message{
            Key:  []byte("Atraccion " + s.IdAtraccion),
            Value: []byte(s.IdAtraccion + ":" + strconv.Itoa(s.Personas)),
        })
    if err != nil {
        panic("Error al conectarse al gestor de colas - No se puede mandar la información al servidor de tiempos de espera: " + err.Error())
    }

    // Generamos un número aleatorio de personas en cola
    rand.Seed(time.Now().UnixNano()) // Utilizamos la función Seed(semilla) para inicializar la fuente predeterminada al requerir un comportamiento reproducible
    min := 0
    max := 60
    s.Personas = (rand.Intn(max-min+1) + min)

    // Cada 1 a 3 segundos el sensor envía la información al servidor de tiempos
    time.Sleep(time.Duration(tiempoAleatorio) * time.Second)

    fmt.Println("En la atracción [" + s.IdAtraccion + "] hay " + strconv.Itoa(s.Personas) + " personas en cola")
}
}
```

Continuando con la función, en esta vamos a generar un productor de Kafka que manda la información a la dirección indicada en la directiva “Brokers” a través del Topic “sensor-servidorTiempos”.

Para entonces abrir un bucle for infinito que continuamente mandará la información del sensor al servidor de tiempos. En este bucle por medio del productor de Kafka que hemos

generado indicamos la Key y el Value que envía un array de bytes donde ponemos el id de la atracción y el número de personas en cola de dicha atracción.

Controlamos además por si se produce un error al escribir el mensaje por el Kafka, que se nos muestre un mensaje por consola de información al respecto.

Y luego generamos de nuevo el número de personas en cola para que el servidor de tiempos de espera vaya actualizando sus tiempos de espera en base a las colas actuales.

Finalmente hacemos un time.Sleep con el tiempo aleatorio que generamos previamente para que el bucle se vuelva a ejecutar pasado dicho tiempo.

Para terminar con el microservicio tenemos la función de creación topics:

```
/*
 * Función que crea el topic para el envío de los movimientos de los visitantes
 */
func crearTopic(IPBroker, PuertoBroker string) {

    topic := "sensor-servidorTiempos"
    conn, err := kafka.Dial("tcp", IPBroker+":"+PuertoBroker)
    if err != nil {
        panic(err.Error())
    }
    defer conn.Close()

    controller, err := conn.Controller()

    if err != nil {
        panic(err.Error())
    }
    //Creamos una variable del tipo kafka.Conn
    var controllerConn *kafka.Conn
    //Le damos los valores necesarios para crear el controllerConn
    controllerConn, err = kafka.Dial("tcp", net.JoinHostPort(controller.Host, strconv.Itoa(controller.Port)))
    if err != nil {
        panic(err.Error())
    }
    defer controllerConn.Close()
    //Configuración del topic mapa-visitantes
    topicConfigs := []kafka.TopicConfig{
        kafka.TopicConfig{      redundant type from array, slice, or map composite literal
            Topic:          topic,
            NumPartitions: 1,
            ReplicationFactor: 1,
        },
    }
    err = controllerConn.CreateTopics(topicConfigs...)
    if err != nil {
        panic(err.Error())
    }
}
```

En esta empezamos indicando el topic a crear, preparando el dial y asegurándonos de cerrar dicha conexión.

Luego creamos y configuramos el controller que utilizamos a posteriori para la creación del topic que hemos indicado, al cual le podemos personalizar el número de particiones y el valor de replication factor.

Fwq_waitingTimeServer

Continuamos con el servidor de tiempos de espera, el cual implementa la funcionalidad necesaria para conocer el tiempo de espera de cada atracción. Este microservicio recibe a través del gestor de colas, por parte de los sensores, el número de visitantes que hay en las colas de cada atracción. Y además, permanece a la escucha indefinidamente esperando a que el Engine le pida los tiempos de espera de todas las atracciones.

```
100% (0:00:00.000) 0 authors (0:00:00.000)
package main

import (
    "bufio"
    "context"
    "crypto/rand"
    "fmt"
    "log"
    "net"
    "os"
    "strconv"
    "strings"
    "time"

    _ "github.com/go-sql-driver/mysql"
    "github.com/oklog/ulid"
    "github.com/segmentio/kafka-go"
)

/*
 * Estructura de las atracciones
 */
You, 4 days ago | 1 author (You)
type atraccion struct {
    ID      string `json:"id"`
    TCiclo  int    `json:"tciclo"`
    NVisitantes int   `json:"nvisitantes"`
    Posicionx int    `json:"posicionx"`
    Posiciony int    `json:"posiciony"`
    TiempoEspera int   `json:"tiempoEspera"`
    Estado   string `json:"estado"`
    Parque   string `json:"parqueAtracciones"`
}
You, 7 months ago • Sensores en principio terminados y servido
}

var atracciones []atraccion // Variable global que irá almacenando la información actual de las atracciones del parque
```

Como siempre, declaramos el paquete, las librerías importadas y además de esto, generamos un struct “atraccion” para poder gestionar más fácilmente la información de las atracciones. Adicionalmente creamos una variable global para almacenar la información en tiempo real de las atracciones del parque, esto lo hacemos porque este módulo no puede acceder a la BD para consultar dicha información y por lo tanto depende de que el engine se la proporcione, por eso la almacenamos con esta variable.

```

func main() {
    host := os.Args[1]
    puertoEscucha := os.Args[2]
    ipBrokerGestorColas := os.Args[3]
    puertoBrokerGestorColas := os.Args[4]

    crearTopic(ipBrokerGestorColas, puertoBrokerGestorColas)

    // Arrancamos el servidor y atendemos conexiones entrantes
    fmt.Println("Servidor de tiempos atendiendo en " + host + ":" + puertoEscucha)
    fmt.Println() // Por limpieza

    l, err := net.Listen("tcp", host+":"+puertoEscucha)

    if err != nil {
        fmt.Println("Error escuchando", err.Error())
        os.Exit(1)
    }

    // Cerramos el listener cuando se cierra la aplicación
    defer l.Close()

    go recibirInfoSensores(ipBrokerGestorColas, puertoBrokerGestorColas)

    // Bucle infinito hasta la salida del programa en el que se tratan las peticiones recibidas por el engine
    for {

        // Atendemos conexiones entrantes
        c, err := l.Accept()
        if err != nil {
            fmt.Println("Error conectando con el engine:", err.Error())
        }

        // Imprimimos la dirección de conexión del cliente
        fmt.Println() // Por limpieza
        log.Println("Cliente engine " + c.RemoteAddr().String() + " conectado.\n")

        // Manejamos las conexiones de forma concurrente
        go manejoConexion(c)
    }
}

```

En el main empezamos guardando la información pasada por parámetro donde tenemos la ip y puerto de escucha del servidor de tiempos y la ip y el puerto del broker gestor de colas. A continuación generamos el topic a utilizar para la recepción de la información por parte de los sensores, para después escuchar las peticiones por sockets enviadas por el engine con los tiempos de espera de las atracciones actuales.

Seguidamente tenemos una llamada a una go routine en la función “recibirInfoSensores” que va a gestionar la recepción de las personas que se encuentran en las colas de las atracciones.

Y ya entrando en lo importante, hacemos una llamada asíncrona y concurrente a la función manejoConexion que es la función que se va a encargar de recibir la información de los sensores por medio de un consumidor Kafka. Dicha llamada está englobada en un bucle for en el que aceptamos las conexiones entrantes, es decir, las peticiones del engine para que le envíemos los tiempos de espera de las atracciones actualizados.

Centrándonos en el código de las funciones vamos con la más irrelevante que es la de creación de topics que al seguir la misma mecánica que la creada en el módulo de los sensores no vamos a entrar en detalle.

```
/*
 * Función que crea el topic para el envío de los movimientos de los visitantes
 */
func crearTopic(IpBroker, PuertoBroker string) {

    topic := "sensor-servidorTiempos"
    conn, err := kafka.Dial("tcp", IpBroker+":"+PuertoBroker)
    if err != nil {
        panic(err.Error())
    }
    defer conn.Close()

    controller, err := conn.Controller()
    if err != nil {
        panic(err.Error())
    }
    //Creamos una variable del tipo kafka.Conn
    var controllerConn *kafka.Conn
    //Le damos los valores necesarios para crear el controllerConn
    controllerConn, err = kafka.Dial("tcp", net.JoinHostPort(controller.Host, strconv.Itoa(controller.Port)))
    if err != nil {
        panic(err.Error())
    }
    defer controllerConn.Close()
    //Configuración del topic mapa-visitantes
    topicConfigs := []kafka.TopicConfig{
        kafka.TopicConfig{ Topic: topic,
                           NumPartitions: 1,
                           ReplicationFactor: 1,
                         },
    }
    err = controllerConn.CreateTopics(topicConfigs...)
    if err != nil {
        panic(err.Error())
    }
}

func Ulid() string {
    t := time.Now().UTC()
    id := ulid.MustNew(ulid.Timestamp(t), hho.Reader)

    return id.String()
}
```

La siguiente función que vamos a comentar y que si que ya tiene relevancia es la antes mencionada “recibirInfoSensores”:

```
/* Función que se encarga de recibir el número de personas que hay en las colas por parte de los sensores */
func recibirInfoSensores(IpBroker, PuertoBroker string) {

    broker := IpBroker + ":" + PuertoBroker
    r := kafka.ReaderConfig(kafka.ReaderConfig{
        Brokers:    []string{broker},
        Topic:      "sensor-servidorTiempos",
        GroupID:    Ulid(),
        StartOffset: kafka.LastOffset,
    })

    reader := kafka.NewReader(r)

    for {

        m, err := reader.ReadMessage(context.Background())
        if err != nil {
            fmt.Println("Ha ocurrido algún error a la hora de conectarse con kafka", err)
        }

        if strings.Contains(string(m.Value), "desconectado") {
            log.Println(string(m.Value))
        } else {
            fmt.Println("[", string(m.Value)+" personas en cola", "]")

            infoSensor := strings.Split(string(m.Value), ":")

            idAtraccion := infoSensor[0]
            personasEnCola, _ := strconv.Atoi(infoSensor[1])

            encontrado := false

            // Buscamos la atracción indicada por el sensor para calcular su tiempo de espera actual
            for i := 0; i < len(atracciones) && !encontrado; i++ {

                if atracciones[i].ID == idAtraccion {
                    encontrado = true
                    atracciones[i].TiempoEspera = calculaTiempoEspera(atracciones[i], personasEnCola)
                }
            }
        }
    }
}
```

En esta función tenemos implementado un consumidor de kafka que como curiosidad utiliza una función auxiliar llamada “Ulid” que nos sirve para generar GroupID aleatorios para recibir la info de los sensores, y tras esto un bucle for infinito para procesar de manera indefinida las comunicaciones entrantes de los sensores. En dicho bucle for procesamos dos tipos de mensajes, los que nos informan de la caída de los sensores y los que contienen el número de personas en la cola de una atracción. Entonces, si se trata del segundo caso, buscamos el id de la atracción indicada por el sensor para actualizar el tiempo de espera en base al número de personas en la cola.

Y dicho esto vamos a hablar ahora de la función que realiza el cálculo de los nuevos tiempos de espera:

```
/* Función que calcula el tiempo de espera de una atracción dada una cantidad de personas en la cola */
func calculaTiempoEspera(a atraccion, personasEnCola int) int {

    tiempoEspera := 0

    // Mientras el número de personas en la cola sea mayor o igual al número de personas que pueden subir a la atracción
    for personasEnCola >= a.NVisitantes {

        tiempoEspera += a.TCiclo
        personasEnCola -= a.NVisitantes

    }

    return tiempoEspera
}
```

Esta función se basa en el número de personas en cola y el número de visitantes que pueden montarse en una atracción en un único ciclo, por lo que mientras las personas en cola sean mayores o iguales a las personas que pueden subir a la atracción, vamos a ir incrementando el tiempo de espera sumando el tiempo de ciclo en cada iteración.

Y por último, nos queda la función más importante, que la llamada “manejoConexion” que se encarga de gestionar la lógica para cada petición de conexión recibida desde el engine.

```
// Función que maneja la lógica para una única petición de conexión
func manejoConexion(conn net.Conn) {

    // Lectura del buffer de entrada hasta el final de línea
    buffer, err := bufio.NewReader(conn).ReadBytes('\n')

    // Cerrar las conexiones con engines desconectados
    if err != nil {
        log.Println("Engine" + conn.RemoteAddr().String() + " desconectado.\n")
        conn.Close()
        return
    }

    //fmt.Println("Petición del Engine: " + string(buffer))
    log.Println("Petición de un engine recibida.")

    infoAtracciones := strings.Split(string(buffer), " | ")
```

Empezamos teniendo en cuenta que las peticiones recibidas del engine contienen los tiempos de espera actuales de todas y cada una de las atracciones. Y es por esto que con el split guardamos dicha información en un array. Por otro lado, nos aseguramos de cerrar las conexiones con engines desconectados.

```

fmt.Println() // Por limpieza
fmt.Println("Estado actual de las atracciones:")
fmt.Println("ID:TCiclo:NVisitantes:TiempoEspera")
// El -1 es porque la longitud del array es 17, ya que al terminar la cadena en la barra vertical
// el split realizado coge un último elemento para formar el array que contiene un espacio en blanco.
for i := 0; i < (len(infoAtracciones) - 1); i++ {

    fmt.Println(infoAtracciones[i])
    infoAtraccion := strings.Split(infoAtracciones[i], ":")

    var a = atraccion{
        ID:          "",
        TCiclo:      -1,
        NVisitantes: -1,
        TiempoEspera: -1,
    }

    a.ID = infoAtraccion[0]
    tciclo, _ := strconv.Atoi(infoAtraccion[1])
    a.TCiclo = tciclo
    nvisitantes, _ := strconv.Atoi(infoAtraccion[2])
    a.NVisitantes = nvisitantes
    tiempoEspera, _ := strconv.Atoi(infoAtraccion[3])
    a.TiempoEspera = tiempoEspera

    encontrada := false

    // Si la atracción ya la tenemos almacenada, actualizamos su información y sino la añadimos al slice
    for i := 0; i < len(atracciones) && !encontrada; i++ {
        if atracciones[i].ID == a.ID {
            encontrada = true
        }
    }

    if !encontrada {
        atracciones = append(atracciones, a)
    }
}

fmt.Println() // Por limpieza

```

En este bucle for simplemente procesamos la información almacenada en el array mencionado antes y guardamos todas las atracciones en la variable global.

```

// Enviamos al engine los tiempos de espera actuales
tiemposEspera := ""

// Formamos la cadena con los tiempos de espera que le vamos a mandar al engine
for i := 0; i < len(atracciones); i++ {
    tiemposEspera += atracciones[i].ID + ":" + strconv.Itoa(atracciones[i].TiempoEspera) + "|"
}

// Mandamos una cadena separada por barras con los tiempos de espera de cada atracción al engine
conn.Write([]byte(tiemposEspera))
fmt.Println("Enviendo los tiempos de espera actualizados...")
fmt.Println() // Por limpieza
conn.Close() // Cerramos la conexión con el engine

manejoConexion(conn) // Reiniciamos el proceso
}

```

You, 7 months ago • Envio de información por parte del sensor adaptad...

Y por último, generamos la cadena de respuesta que vamos a enviar al engine dónde ya tendremos los tiempos de espera actualizados, ya que de eso ya se encargó la otra función

concurrente que recibía la información de los sensores, y es por esto el uso de la variable global. Tras mandar la respuesta, hacemos la llamada recursiva a esta función para la correcta gestión de las conexiones vía socket y así cerrar aquellas que corresponda.

Fwq_visitor

Los visitantes son las personas que entrarán al parque de atracciones, primero tendrán que registrarse. En este caso solo tienen que introducir su ID, nombre y contraseña.

Después tendrán una opción de edición de perfil donde podrán cambiar alguna información. Después tendrá la opción de moverse por el parque de atracciones, el cual se irá moviendo a las atracciones que tengan un tiempo de espera menor de 60 minutos y finalmente podrán salir del parque de atracciones.

Los visitantes cuentan con tres topics. Uno por el cual irán enviando las peticiones al engine, otro por el que leen la respuesta del engine a la petición de login y otro por donde recibirán el mapa por parte del engine.

De forma adicional, el visitante tiene la opción de poder registrarse y modificar su perfil conectándose al registry vía sockets o por api rest, esto será elegible antes de cada una de estas operaciones.

Ahora comentaré algunas características principales del código.

```
package main

import (
    "bufio"
    "bytes"
    "context"
    "crypto/aes"
    "crypto/cipher"
    "crypto/rand"
    "crypto/tls"
    "encoding/base64"
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "math"
    "math/rand"
    "net"
    "net/http"
    "os"
    "os/signal"
    "runtime/pprof"
    "strconv"
    "strings"
    "time"

    "github.com/oklog/ulid/v2"
    "github.com/segmentio/kafka-go"
)
```

Primero empezamos definiendo la paqueterías y librerías necesarias.

```
  "github.com/oklog/ulid/v2"
  "github.com/segmentio/kafka-go"
)

/*
 * Estructura del visitante
 */
You, 2 weeks ago | 1 author (You)
type visitante struct {
    ID          string `json:"id"`
    Nombre      string `json:"nombre"`
    Password    string `json:"contraseña"`
    Posicionx   int    `json:"posicionx"`
    Posiciony   int    `json:"posiciony"`
    Destinox    int    `json:"destinox"`
    Destinoy    int    `json:"destinoy"`
    DentroParque int   `json:"dentroParque"`
    IdEnParque  string `json:"idParque"`
    UltimoEvento string `json:"ultimoEvento"`
    Parque      string `json:"parqueAtracciones"`
}

/*
 * Estructura de las atracciones
 */
You, 5 days ago | 1 author (You)
type atraccion struct {
    ID          string `json:"id"`
    TCiclo      int    `json:"tciclo"`
    NVisitantes int   `json:"nvisitantes"`
    Posicionx   int    `json:"posicionx"`
    Posiciony   int    `json:"posiciony"`
    TiempoEspera int   `json:"tiempoEspera"`
    Estado      string `json:"estado"`
    Parque      string `json:"parqueAtracciones"`
}

const (
    connType = "tcp"
)
```

Luego definimos los structs visitante y atracción junto con la constante connType que indica el tipo de conexión que va a utilizar el socket para conectarnos al registry.

```

var v = visitante{ // Guardamos la información del visitante que nos hace falta
    ID:      "",
    Password:  "",
    Posicionx: 0,
    Posiciony: 0,
    Destinox: -1,
    Destinoy: -1,
    DentroParque: 0,
}

var a = atraccion{ // Guardamos la información de la atraccion que nos hace falta
    Posicionx: -1,
    Posiciony: -1,
    TiempoEspera: -1,
}

// Inicializamos un vector con bytes aleatorios
var iv = []byte{35, 46, 57, 24, 85, 35, 24, 74, 87, 35, 88, 98, 66, 32, 14, 05}

```

Ahora nos declaramos un par de variables globales para gestionar la información del visitante y de la atracción destino de dicho visitante y además añadimos la inicialización de un vector con cantidades de bytes aleatorias para el cifrado AES que vamos a usar para la securización del kafka entre el visitante y el engine.

```

/**
 * Función main de los visitantes
 */
func main() {

    //Argumentos iniciales
    IpFWQ_Registry := os.Args[1]
    PuertoRegistrySockets := os.Args[2]
    PuertoRegistryApiRest := os.Args[3]
    IpBroker := os.Args[4]
    PuertoBroker := os.Args[5]
    crearTopic(IpBroker, PuertoBroker, "peticiones")
    crearTopic(IpBroker, PuertoBroker, "respuesta-login")
    crearTopic(IpBroker, PuertoBroker, "movimiento-mapa")

    fmt.Println("Creado un visitante que envía peticiones a un registry por " + IpFWQ_Registry + ":"
    fmt.Println() // Por limpieza

    fmt.Println("***Bienvenido al parque de atracciones***")
    fmt.Println()

    MenuParque(IpFWQ_Registry, PuertoRegistrySockets, PuertoRegistryApiRest, IpBroker, PuertoBroker)

}

```

Este es el main de la aplicación visitante, donde primero capturamos la información pasada por parámetro, luego generamos los 3 topics necesarios y por último llamamos a la función MenuParque que nos va a mostrar el menú de la aplicación.

```

/*
 * Función que pinta el menú del parque
 */
func MenuParque(IpFWQ_Registry, PuertoRegistrySockets, PuertoRegistryApiRest, IpBroker, PuertoBroker string) {
    var opcion int
    //Guardamos la opción elegida
    for {
        fmt.Println("****Menu parque de atracciones****")
        fmt.Println("1.Crear perfil")
        fmt.Println("2.Editar perfil")
        fmt.Println("3.Moverse por el parque")
        //fmt.Println("4.Salir del parque")
        fmt.Print("Elige la acción a realizar:")
        fmt.Scanln(&opcion)

        switch os := opcion; os {
        case 1:
            CrearPerfil(IpFWQ_Registry, PuertoRegistrySockets, PuertoRegistryApiRest)
        case 2:
            EditarPerfil(IpFWQ_Registry, PuertoRegistrySockets, PuertoRegistryApiRest)
        case 3:
            EntradaParque(IpBroker, PuertoBroker)
        default:
            fmt.Println("Opción invalida, elige otra opción")
        }
    }
}

```

Y entrando en las funciones que se pueden ver en el menú, vamos con la primera que nos sirve para la creación de perfiles de visitantes del parque.

```

/* Función que se conecta al módulo FWQ_Registry para crear un nuevo usuario */
func CrearPerfil(ipRegistry, puertoRegistrySockets, puertoRegistryApiRest string) {

    fmt.Println() // Por limpieza
    fmt.Println("*****Creación de perfil*****")
    fmt.Println() // Por limpieza

    // Damos la posibilidad elegir conexión vía sockets o por API_REST
    fmt.Println("Selecciona el tipo de conexión al registry:")
    fmt.Println("1 -> Sockets")
    fmt.Println("2 -> API REST")
    fmt.Println() // Por limpieza

    var eleccion int
    fmt.Scan(&eleccion)
    fmt.Println() // Por limpieza
}

```

En esta función empezamos dando la opción requerida para la entrega de julio, y es que el usuario tenga la opción de elegir si quiere emplear la comunicación vía sockets o vía api rest.

```

// Si el usuario elige la conexión por sockets
if elección == 1 {

    cert, err := tls.LoadX509KeyPair("cert/cert.pem", "cert/key.pem")
    if err != nil {
        log.Fatal("Error al cargar los ficheros de certificado y clave asociada: ", err)
    }

    config := tls.Config{
        Certificates: []tls.Certificate{cert},
        InsecureSkipVerify: true, // Para que admita certificados autofirmados
    }

    //conn, err := net.Dial(connType, ipRegistry+":"+puertoRegistry) CONEXIÓN INSEGURA
    conn, err := tls.Dial(connType, ipRegistry+":"+puertoRegistrySockets, &config) // CONEXIÓN SEGURA
}

```

En caso de elegir la conexión por sockets, empezamos cargando el certificado autofirmado para utilizarlo en la configuración TLS que van a usar los sockets, indicando en dicha configuración que admitimos certificados autofirmados. Tras esto establecemos la conexión segura vía socket con el registry.

```

if err != nil {
    fmt.Println("Error al conectarse al Registry:", err.Error())
} else { // Si el visitante establece conexión con el Registry indicado por parámetro

    defer conn.Close() // Nos aseguramos de cerrar la conexión

    conn.Write([]byte("1" + "\n")) // Le pasamos al Registry la opción elegida por el visitante

    reader := bufio.NewReader(os.Stdin)

    fmt.Print("Introduce tu ID:")
    id, _ := reader.ReadString('\n')

    // Nos aseguramos de que no sea válido un id en blanco
    if len(id) > 1 {

        conn.Write([]byte(id))

        fmt.Print("Introduce tu nombre:")
        nombre, _ := reader.ReadString('\n')

        // Nos aseguramos de que no sea válido un nombre en blanco
        if len(nombre) > 1 {

            conn.Write([]byte(nombre))

            fmt.Print("Introduce tu contraseña:")
            password, _ := reader.ReadString('\n')

            // Nos aseguramos de que no sea válida una contraseña en blanco
            if len(password) > 1 {

                conn.Write([]byte(password))

                //Escuchando por el relay el mensaje de respuesta del Registry
                message, _ := bufio.NewReader(conn).ReadString('\n')

                // Comprobamos si el Registry nos devuelve un mensaje de respuesta
                if message != "" {
                    log.Println("Respuesta del Registry: ", message)
                } else {
                    log.Println("Lo siento, el Registry no está disponible en estos momentos.")
                }

                } else {
                    fmt.Println("ERROR: Por favor introduzca una contraseña que no sea vacía.")
                }

            } else {
                fmt.Println("ERROR: Por favor introduzca un nombre que no sea vacío.")
            }

        } else {
            fmt.Println("ERROR: Por favor introduzca un ID que no sea vacío.")
        }
    }
}

```

Y tras tratar de establecer la conexión se pueden dar 2 casuísticas, que el registry no esté disponible y por lo tanto no podamos establecer la conexión, o que si se encuentra disponible el registry y por ello le indicamos que el visitante ha solicitado un registro, en cuyo caso se nos va a solicitar introducir un id, nombre y contraseña no vacíos para mandarlos al registry y proceder con el registro del visitante. Si por lo que fuera el registry se cae durante la conexión, no recibiremos un mensaje de respuesta y lo notificamos al usuario por la consola.

```
 } else if eleccion == 2 { // Si el usuario elige la conexión por API_REST
    var id string
    fmt.Println("Introduce tu ID:")
    fmt.Scanln(&id)

    // Nos aseguramos de que no sea válido un id en blanco
    if len(id) > 1 {

        var nombre string
        fmt.Println("Introduce tu nombre:")
        fmt.Scanln(&nombre)

        // Nos aseguramos de que no sea válido un nombre en blanco
        if len(nombre) > 1 {

            var password string
            fmt.Println("Introduce tu contraseña:")
            fmt.Scanln(&password)

            // Nos aseguramos de que no sea válida una contraseña en blanco
            if len(password) > 1 {

                v := visitante{
                    ID:      id,
                    Nombre:  nombre,
                    Password: password,
                }
                vComoJson, err := json.Marshal(v)
                if err != nil {
                    log.Fatalf("Error codificando visitante como JSON: %v", err)
                }

                // Realizamos la composición de los datos
                datos := strings.NewReader(string(vComoJson))

                tr := &http.Transport{
                    TLSClientConfig: &tls.Config{InsecureSkipVerify: true}, // Para que admita certificados autofirmados
                }
                http := &http.Client{Transport: tr}
```

La otra opción sería que se elija el registro del visitante en el registry por medio de api rest, en cuyo caso, al igual que con los sockets se deberán introducir un id, nombre y contraseña no vacíos y tras esto nos generamos una variable de tipo struct visitante para almacenar la información que hemos introducido, convertirla a json para su posterior envío, realizando además la composición de los datos a enviar. Y por último, en esta captura tenemos la configuración TLS para el cliente http que vamos a usar para establecer la comunicación vía https, donde indicamos que se admiten certificados autofirmados.

```

    // Ahora realizamos el envío de los datos
    res, err := http.Post("https://"+ipRegistry+":"+puertoRegistryApiRest+"/crear/"+v.ID, "application/json", datos)
    if err != nil {
        fmt.Printf("Error al realizar la petición al servidor API REST: %v\n", err)
        return
    }

    // Nos aseguramos de que se cierra el body
    defer res.Body.Close()

    // Realizamos la lectura del body
    body, err := ioutil.ReadAll(res.Body)
    if err != nil {
        fmt.Printf("Error al leer la respuesta recibida: %v\n", err)
        return
    }

    fmt.Println() // Por limpieza
    fmt.Printf("%s", body)
    fmt.Println() // Por limpieza

} else {
    fmt.Println("ERROR: Por favor introduzca una contraseña que no sea vacía.")
}

} else {
    fmt.Println("ERROR: Por favor introduzca un nombre que no sea vacío.")
}

} else {
    fmt.Println("ERROR: Por favor introduzca un ID que no sea vacio.")
}

} else { // Si la opción introducida no es válida
    fmt.Println("ERROR: Por favor introduzca 1 o 2")
    fmt.Println() // Por limpieza
}

```

Ahora se realiza el envío de la petición POST en este caso, donde pasamos los datos antes mencionados, y realizamos la posterior lectura del body de la respuesta del servidor a nuestra petición, la cual imprimimos por pantalla.

Ahora veremos la edición de perfil que es muy similar al registro.

```
/* Función que se conecta al módulo FWQ_Registry para editar o actualizar el perfil de un usuario existente */
func EditarPerfil(ipRegistry, puertoRegistrySockets, puertoRegistryApiRest string) {

    fmt.Println() // Por limpieza
    fmt.Println("*****Modificación de perfil*****")
    fmt.Println() // Por limpieza

    // Damos la posibilidad elegir conexión vía sockets o por API_REST
    fmt.Println("Selecciona el tipo de conexión al registry:")
    fmt.Println("1 -> Sockets")
    fmt.Println("2 -> API_REST")
    fmt.Println() // Por limpieza

    var eleccion int
    fmt.Scan(&eleccion)
    fmt.Println() // Por limpieza

    // Si el usuario elige la conexión por sockets
    if eleccion == 1 {

        cert, err := tls.LoadX509KeyPair("cert/cert.pem", "cert/key.pem")
        if err != nil {
            log.Fatal("Error al cargar los ficheros de certificado y clave asociada: ", err)
        }

        config := tls.Config{
            Certificates: []tls.Certificate{cert},
            InsecureSkipVerify: true, // Para que admita certificados autofirmados
        }

        // conn, err := net.Dial(connType, ipRegistry+":"+puertoRegistry) CONEXIÓN INSEGURA
        conn, err := tls.Dial(connType, ipRegistry+":"+puertoRegistrySockets, &config) // CONEXIÓN SEGURA
    }
}
```

Como podemos ver le pasamos como parámetros la ip y los puertos del registry al igual que en el caso de la función de registro.

Y realmente el código es muy similar al caso anterior, sólo que ahora envíamos la opción 2 en lugar de la opción 1 para que el Registry sepa que lo que queremos es un update de nuestros datos.

```

if err != nil {
    fmt.Println("Error al conectarse al Registry:", err.Error())
} else { // Si el visitante establece conexión con el Registry indicado por parámetro

    defer conn.Close() // Nos aseguramos de cerrar la conexión

    conn.Write([]byte("2" + "\n")) // Le pasamos al Registry la opción elegida por el visitante

    reader := bufio.NewReader(os.Stdin)

    fmt.Println("Información del visitante que se quiere modificar:")
    fmt.Print("Introduce el ID:")
    id, _ := reader.ReadString('\n')

    // Nos aseguramos de que el ID no sea vacío.
    if len(id) > 1 {

        conn.Write([]byte(id))

        fmt.Print("Introduce el nombre:")
        nombre, _ := reader.ReadString('\n')

        // Nos aseguramos de que el nombre no sea vacío.
        if len(nombre) > 1 {

            conn.Write([]byte(nombre))

            fmt.Print("Introduce la contraseña:")
            password, _ := reader.ReadString('\n')

            // Nos aseguramos de que la contraseña no sea vacía.
            if len(password) > 1 {

                conn.Write([]byte(password))

                message, _ := bufio.NewReader(conn).ReadString('\n')

                // Comprobamos si el Registry nos devuelve un mensaje de respuesta
                if message != "" {
                    log.Println("Respuesta del Registry: ", message)
                } else {
                    log.Println("Lo siento, el Registry no está disponible en estos momentos.")
                }

            } else {
                fmt.Println("ERROR: Por favor introduzca una contraseña que no sea vacía.")
            }
        }
    }
}

```

```

    } else {
        fmt.Println("ERROR: Por favor introduzca un nombre que no sea vacío.")
    }

} else {
    fmt.Println("ERROR: Por favor introduzca un ID que no sea vacío.")
}

}

} else if eleccion == 2 { // Si el usuario elige la conexión por API_REST

    var id string

    fmt.Print("Introduce tu ID:")
    fmt.Scanln(&id)

    // Nos aseguramos de que no sea válido un id en blanco
    if len(id) > 1 {

        var nombre string

        fmt.Print("Introduce tu nombre:")
        fmt.Scanln(&nombre)

        // Nos aseguramos de que no sea válido un nombre en blanco
        if len(nombre) > 1 {

            var password string

            fmt.Print("Introduce tu contraseña:")
            fmt.Scanln(&password)

            // Nos aseguramos de que no sea válida una contraseña en blanco
            if len(password) > 1 {

                tr := &http.Transport{
                    TLSClientConfig: &tls.Config{InsecureSkipVerify: true}, // Para que admita certificados autofirmados
                }

                // Accedemos al cliente mediante http.Client
                clienteHttp := &http.Client{Transport: tr}

                v := visitante{
                    ID:      id,
                    Nombre: nombre,
                    Password: password,
                }
            }
        }
    }
}

```

```

vComoJson, err := json.Marshal(v)
if err != nil {
    log.Fatalf("Error codificando visitante como JSON: %v", err)
}

// Creamos una nueva petición tipo PUT mediante http.NewRequest
peticion, err := http.NewRequest("PUT", "https://" + ipRegistry + ":" + puertoRegistryApiRest + "/editar/" + v.ID, bytes.NewBuffer(vComoJson))
if err != nil {
    log.Fatalf("Error creando la petición PUT: %v", err)
}

// Agregamos las cabeceras que queramos
peticion.Header.Add("Content-Type", "application/json")

respuesta, err := clienteHttp.Do(peticion)
if err != nil {
    fmt.Printf("Error al realizar la petición al servidor API REST: %v\n", err)
    return
}

// Nos aseguramos de que se cierra el body recibido
defer respuesta.Body.Close()

// Realizamos la lectura del body
cuerpoRespuesta, err := ioutil.ReadAll(respuesta.Body)
if err != nil {
    fmt.Printf("Error al leer la respuesta recibida: %v\n", err)
    return
}

// Aquí podemos decodificar la respuesta si es un JSON, o convertirla a cadena
fmt.Println() // Por limpieza
fmt.Println("%s", cuerpoRespuesta)
fmt.Println() // Por limpieza

} else {
    fmt.Println("ERROR: Por favor introduzca una contraseña que no sea vacía.")
}

} else {
    fmt.Println("ERROR: Por favor introduzca un nombre que no sea vacío.")
}

} else {
    fmt.Println("ERROR: Por favor introduzca un ID que no sea vacío.")
}

} else { // Si la opción introducida no es válida
    fmt.Println("ERROR: Por favor introduzca 1 o 2")
    fmt.Println() // Por limpieza
}

```

Como vemos la forma de proceder es la misma, salvo que para la petición PUT creamos un cliente http en concreto y tenemos que crear la petición y después ejecutarla con la función Do.

Con relación a la entrada al parque por parte de los visitantes tenemos el siguiente código.

```
/* Función que envía las credenciales de acceso del visitante para entrar en el parque */
func EntradaParque(IpBroker, PuertoBroker string) {
    fmt.Println("Bienvenido al parque de atracciones")

    reader := bufio.NewReader(os.Stdin)

    fmt.Print("Por favor introduce tu alias:")
    alias, _ := reader.ReadString('\n')

    if len(alias) > 1 {

        v.ID += strings.TrimSpace(string(alias))

        fmt.Print("y tu password:")
        password, _ := reader.ReadString('\n')

        if len(password) > 1 {

            v.Password += strings.TrimSpace(string(password))

            // SECURIZAMOS LA COMUNICACIÓN EN KAFKA
            // Cargamos la clave de cifrado AES del archivo
            fichero, err := ioutil.ReadFile("claveCifradoAES.txt")
            if err != nil {
                log.Fatal("Error al leer el archivo de la clave de cifrado AES: ", err)
            }

            clave := string(fichero) // Clave de 24 bits

            // Preparamos las credenciales de inicio de sesión del visitante
            mensaje := strings.TrimSpace(string(alias)) + ":" + strings.TrimSpace(string(password)) + ":" + strconv.Itoa(v.Destinox) + "," + strconv.Itoa(v.Destinoy)

            // Ciframos el mensaje
            mensajeCifrado, err := encriptacionAES(mensaje, clave)
            if err != nil {
                panic(err)
            }

            // Mandamos al engine las credenciales de inicio de sesión del visitante para entrar al parque
            productorLogin(IpBroker, PuertoBroker, mensajeCifrado)
        }
    }
}
```

Donde el visitante introduce su id/alias así como su contraseña comprobando que no sean vacíos y mandará por un topic la información proporcionada cifrada mediante el algoritmo de cifrado simétrico AES, para solicitar la entrada al parque, es decir, el visitante envía una petición de inicio de sesión y para ello tenemos implementada la función productorLogin.

```
/* Función que se encarga de enviar las credenciales de inicio de sesión */
func productorLogin(IpBroker, PuertoBroker, credenciales string, ctx context.Context) {

    var brokerAddress string = IpBroker + ":" + PuertoBroker
    var topic string = "peticiones"

    w := kafka.NewWriter(kafka.WriterConfig{
        Brokers:          []string{brokerAddress},
        Topic:           topic,
        CompressionCodec: kafka.Snappy.Codec(),
    })

    err := w.WriteMessages(ctx, kafka.Message{
        Key:   []byte("Key-Login"),
        Value: []byte(credenciales),
    })
    if err != nil {
        panic("No se pueden encolar las credenciales: " + err.Error())
    }

    fmt.Println("Enviando credenciales -> " + credenciales)
}
```

Esta función implementa un productor de kafka sencillo para enviar las credenciales de inicio de sesión. Dichas credenciales previamente las hemos cifrado, utilizando para ello la función de encriptación AES:

```
/* Función de encriptación que utiliza el algoritmo AES */
func encriptacionAES(texto, claveSecreta string) (string, error) {
    bloqueDeCifrado, err := aes.NewCipher([]byte(claveSecreta)) // Creamos un nuevo bloque de cifrado AES
    if err != nil {
        return "", err
    }

    textoPlano := []byte(texto)
    cfb := cipher.NewCFBEncrypter(bloqueDeCifrado, iv) // Creamos el stream de encriptación
    textoCifrado := make([]byte, len(textoPlano))
    cfb.XORKeyStream(textoCifrado, textoPlano) // Sustituye cada byte de textoPlano por cada byte en el stream de bytes cifrado (textoCifrado)

    return encodeBase64(textoCifrado), nil
}
```

En esta función creamos un nuevo bloque de cifrado AES, después el stream de encriptación y sustituímos cada byte del texto plano por cada byte de dicho stream para devolver el texto cifrado codificado en base 64, para lo que utilizamos esta función auxiliar:

```
/* Función que nos simplifica la llamada de la función EncodeToString en base 64 */
func encodeBase64(src []byte) string {
    return base64.StdEncoding.EncodeToString(src)
}
```

Siguiendo con la función de entrada al parque, a continuación del envío de la petición de login tenemos declarada una función anónima y asíncrona que sirve para que en el momento en el que un visitante se quiera desconectar de la aplicación se envíe una petición de salida:

```
c := make(chan os.Signal, 1)
signal.Notify(c, os.Interrupt)
go func() {
    for sig := range c {
        log.Printf("captured %v, stopping profiler and exiting..", sig)
        mensaje := v.ID + ":" + "OUT" + ":" + strconv.Itoa(v.Destinox) + "," + strconv.Itoa(v.Destinoy)
        mensajeCifrado, err := encriptacionAES(mensaje, clave)
        if err != nil {
            panic(err)
        }
        productorSalir(IpBroker, PuertoBroker, mensajeCifrado)
        fmt.Println()
        fmt.Println("Adios, esperamos que haya disfrutado su estancia en el parque.")
        pprof.StopCPUProfile()
        os.Exit(1)
    }
}()

{()
```

Utilizando para ello la función llamada productorSalir:

```
/* Función que se encarga de mandar la solicitud de salida del parque al engine */
func productorSalir(IpBroker, PuertoBroker, peticion string) {

    var brokerAddress string = IpBroker + ":" + PuertoBroker
    var topic string = "peticiones"

    w := kafka.NewWriter(kafka.WriterConfig{
        Brokers:          []string{brokerAddress},
        Topic:            topic,
        CompressionCodec: kafka.Snappy.Codec(),
        //Dialer:           dialer,
    })

    err := w.WriteMessages(context.Background(), kafka.Message{
        Key:   []byte("Key-Salir"),
        Value: []byte(peticion),
    })
    if err != nil {
        panic("No se puede encolar la solicitud: " + err.Error())
    }

}
```

Y finalizando con la función “entradaParque” tenemos la llamada a “consumidorLogin” para recibir la respuesta del engine a nuestra petición de login.

```
// Recibe del engine el mapa actualizado o un mensaje de parque cerrado
consumidorLogin(IpBroker, PuertoBroker, clave)

} else {
    fmt.Println("ERROR: Por favor introduzca un password no vacío.")
}

} else {
    fmt.Println("ERROR: Por favor introduzca un ID no vacío.")
}

}
```

Centrándonos en la función “consumidorLogin”:

```
/* Función que recibe el mensaje de parque cerrado por parte del engine o no */
func consumidorLogin(IpBroker, PuertoBroker, clave string) {      You, 7 days ago • Im
    respuestaEngine := ""

    broker := IpBroker + ":" + PuertoBroker
    r := kafka.ReaderConfig(kafka.ReaderConfig{
        Brokers: []string{broker},
        Topic:   "respuesta-login",
        GroupID: Ulid(),
        //De esta forma solo cogera los ultimos mensajes despues de unirse al cluster
        StartOffset: kafka.LastOffset,
    })

    reader := kafka.NewReader(r)
    dentroParque := true
```

Creamos un consumidor kafka que configuramos de la siguiente forma. Donde el nombre del topic es respuesta-login, este es el canal por donde recibirá la respuesta a la petición de login del engine. Cabe recalcar que tenemos la opción de StartOffset, la cual hace que cada visitante que se una al tópico empiece a leer desde el último mensaje que ha llegado al topic. Pero un detalle muy relevante respecto a la configuración del consumidor es que utilizamos un GroupID aleatorio, y esto lo hacemos para que cada nuevo proceso de la aplicación visitante posea un GroupID distinto, lo cual nos ha ayudado a resolver conflictos a la hora de tener varios visitantes escuchando de forma simultánea por el mismo engine.

Y para la generación de dichos groupId aleatorios hemos utilizado la siguiente función:

```
func Ulid() string {
    t := time.Now().UTC()
    id := ulid.MustNew(ulid.Timestamp(t), hho.Reader)

    return id.String()
}
```

Por otro lado, respecto a cómo creamos los topics en golang.

Le pasamos por parámetros la ip y puerto por donde kafka va a estar escuchando, lo configuramos conforme nos decía la documentación oficial de kafka en go.

```

/*
 * Función que crea el topic para el envío de los movimientos de los visitantes
 */
func crearTopic(IpBroker, PuertoBroker, topic string) {

    var broker1 string = IpBroker + ":" + PuertoBroker
    conn, err := kafka.Dial("tcp", broker1)
    if err != nil {
        panic(err.Error())
    }
    defer conn.Close()

    controller, err := conn.Controller()

    if err != nil {
        panic(err.Error())
    }
    //Creamos una variable del tipo kafka.Conn
    var controllerConn *kafka.Conn
    //Le damos los valores necesarios para crear el controllerConn
    controllerConn, err = kafka.Dial("tcp", net.JoinHostPort(controller.Host, strconv.Itoa(controller.Port)))
    if err != nil {
        panic(err.Error())
    }
    defer controllerConn.Close()
    //Configuración del topic mapa-visitantes
    topicConfigs := []kafka.TopicConfig{
        kafka.TopicConfig{ Topic: topic,
                           NumPartitions: 10,
                           ReplicationFactor: 1,
        },
    }
    err = controllerConn.CreateTopics(topicConfigs...)
    if err != nil {
        panic(err.Error())
    }
}

```

Como podemos ver, algunos de los valores son similares a la forma en la que creamos los topics como si hiciéramos de forma manual.

```

        for dentroParque {

            m, err := reader.ReadMessage(context.Background())

            if err != nil {
                panic("Ha ocurrido algún error a la hora de conectarse con kafka: " + err.Error())
            }

            respuestaEngine, err = desencriptacionAES(strings.TrimSpace(string(m.Value)), clave)
            if err != nil {
                panic(err)
            }

            log.Println("Respuesta del engine: " + respuestaEngine)

            if respuestaEngine == (v.ID + ":" + "Acceso concedido") {
                v.DentroParque = 1 // El visitante está dentro del parque
                fmt.Println("El visitante está dentro del parque")
                peticionEntrada := v.ID + ":" + "IN" + ":" + strconv.Itoa(v.Destinox) + "," + strconv.Itoa(v.Destinoy)
                peticionEntradaCifrada, err := encriptacionAES(peticionEntrada, clave)
                if err != nil {
                    panic(err)
                }
                productorMovimientos(IpBroker, PuertoBroker, peticionEntradaCifrada) // Le indicamos al engine que el visitante desea entrar al parque
                consumidorMapa(IpBroker, PuertoBroker, clave)
                dentroParque = false
            } else if respuestaEngine == (v.ID + ":" + "Parque cerrado") {
                fmt.Println("Parque cerrado")
                v.DentroParque = 0
                v.ID = ""
                v.Password = ""
                dentroParque = false
            } else if respuestaEngine == (v.ID + ":" + "Aforo al completo") {
                fmt.Println("Aforo al completo")
                v.DentroParque = 0
                v.ID = ""
                v.Password = ""
                dentroParque = false
            }
        }
    }
}

```

Continuando con el código de la función consumidorLogin tenemos un bucle for que hará que se quede escuchando por ese tópico siempre y cuando el visitante se mantenga dentro del parque.

Dentro de dicho bucle hacemos la lectura del mensaje recibido desde el kafka, lo desencriptamos haciendo uso de la función desencriptacionAES:

```

/* Función de desencriptación que utiliza el algoritmo AES */
func desencriptacionAES(texto, claveSecreta string) (string, error) {

    bloqueDeCifrado, err := aes.NewCipher([]byte(claveSecreta)) // Creamos un nuevo bloque de cifrado AES
    if err != nil {
        return "", err
    }

    textoCifrado := decodeBase64(texto)
    cfb := cipher.NewCFBDecrypter(bloqueDeCifrado, iv) // Creamos el stream de desencriptación
    textoPlano := make([]byte, len(textoCifrado))
    cfb.XORKeyStream(textoPlano, textoCifrado) // Sustituye cada byte de textoCifrado por cada byte en textoPlano

    return string(textoPlano), nil
}

```

La cual realiza la operación a la inversa respecto a “encriptacionAES” sirviéndose de una función auxiliar:

```

/* Función que nos simplifica la llamada a DecodeString en base 64 */
func decodeBase64(s string) []byte {
    datos, err := base64.StdEncoding.DecodeString(s)
    if err != nil {
        panic(err)
    }
    return datos
}

```

Para decodificar los mensajes en base 64.

Volviendo a la función “consumidorLogin”, comprobamos si se nos ha concedido el acceso al parque o por el contrario se nos ha denegado, el parque está cerrado o el aforo del parque está completo.

En caso de acceso negativo, simplemente reiniciamos los valores de la variable global que gestiona la información del visitante actual y volvemos al menú principal de la aplicación.

Si por el contrario la solicitud de login ha sido satisfactoria, realizamos la petición de entrada en la que enviamos el movimiento “IN” indicando que vamos a entrar al parque. Dicho movimiento lo enviamos con la función “productorMovimientos”:

```

/* Función que se encarga de enviar los movimientos de los visitantes al engine */
func productorMovimientos(IPBroker, PuertoBroker, movimiento string) {

    var brokerAddress string = IPBroker + ":" + PuertoBroker
    var topic string = "peticiones"

    w := kafka.NewWriter(kafka.WriterConfig{
        Brokers:          []string{brokerAddress},
        Topic:            topic,
        CompressionCodec: kafka.Snappy.Codec(),
    })

    err := w.WriteMessages(context.Background(), kafka.Message{
        Key:   []byte("Key-Moves"),
        Value: []byte(movimiento),
    })
    if err != nil {
        panic("No se puede encolar el movimiento: " + err.Error())
    }

    fmt.Println("Enviando movimiento: " + movimiento)
}

```

Esta función implementa un sencillo productor de kafka que envía los movimientos al engine.

Y volviendo a la función “consumidorLogin”, tras enviar el movimiento de entrada al parque hacemos la llamada a la función “consumidorMapa”:

```
/* Función que recibe el mapa del engine y lo devuelve formateado */
func consumidorMapa(IpBroker, PuertoBroker, clave string) {

    broker := IpBroker + ":" + PuertoBroker
    r := kafka.ReaderConfig(kafka.ReaderConfig{
        Brokers: []string{broker},
        Topic:   "movimiento-mapa",
        GroupID: Ulid(),
        //De esta forma solo cogera los ultimos mensajes despues de unirse al cluster
        StartOffset: kafka.LastOffset,
    })

    reader := kafka.NewReader(r)

    for v.DentroParque == 1 {

        m, err := reader.ReadMessage(context.Background())

        if err != nil {
            panic("Ha ocurrido algún error a la hora de conectarse con kafka: " + err.Error())
        }

        var mapaObtenido string
        err = json.Unmarshal([]byte(string(m.Value)), &mapaObtenido)

        if err != nil {
            fmt.Printf("Error al decodificar el mapa: %v\n", err)
        }

        // Desciframos el mapa
        mapaDescifrado, err := desencriptacionAES(mapaObtenido, clave)
        if err != nil {
            panic(err)
        }
    }
}
```

Esta función implementa un consumidor de kafka que recibe el mapa del engine, lo convierte a string desde json y lo desencripta con nuestra función correspondiente.

```

// Como el parque ha cerrado tenemos que reiniciar la información del visitante
if mapaDescifrado == "Engine no disponible" {
    fmt.Println("El engine ha dejado de estar disponible")
    v.DentroParque = 0
    v.ID = ""
    v.Password = ""
    //v.Posicionx = 0
    //v.Posiciony = 0
    //v.Destinox = -1
    //v.Destinoy = -1
} else {

    // Procesamos el mapa recibido y lo convertimos a un array bidimensional de strings
    //cadenaProcesada := strings.Split(string(m.Value), "|")
    cadenaProcesada := strings.Split(mapaDescifrado, "|")
    var mapa [20][20]string = procesarMapa(cadenaProcesada)
    fmt.Println(mapaDescifrado)
    movimiento := obtenerMovimiento(mapa)
    peticionMovimiento := v.ID + ":" + movimiento + ":" + strconv.Itoa(v.Destinox) + "," + strconv.Itoa(v.Destinoy)
    peticionMovimientoCifrada, err := encriptacionAES(peticionMovimiento, clave)
    if err != nil {
        panic(err)
    }
    productorMovimientos(IpBroker, PuertoBroker, peticionMovimientoCifrada)

    time.Sleep(1 * time.Second) // Mandamos el movimiento del visitante cada segundo| You, now • Uncommitted ch
}

```

Y ahora pueden pasar 2 cosas, que el engine se haya caído, en cuyo caso deja de estar disponible y así no lo ha notificado, por lo que reiniciamos los valores del visitante y volvemos al menú principal de la aplicación.

O que si nos haya enviado el mapa actual del parque en base a nuestro último movimiento enviado y por lo tanto procesamos el mapa con esta función:

```

/* Función que formatea el mapa y lo convierte en un array bidimensional de strings */
func procesarMapa(mapa []string) [20][20]string {
    var mapaFormatado [20][20]string

    k := 0

    for i := 0; i < 20; i++ {
        for j := 0; j < 20; j++ {
            if k < 400 {
                mapaFormatado[i][j] = mapa[k]
                k++
            }
        }
    }

    return mapaFormatado
}

```

Siguiendo con el código de la función consumidorMapa, mostramos el mapa procesado, y ahora que tenemos el mapa disponible nos toca calcular/obtener el movimiento que va a realizar el visitante, para ello llamamos a la función obtenerMovimiento:

```

/* Función que se encarga de ir moviendo al visitante hasta alcanzar el destino */
func obtenerMovimiento(mapa [20][20]string) string {
    var movimiento string

    // Si el visitante no sabe a qué atracción dirigirse o la atracción actual elegida tiene un tiempo de espera mayor a 60 minutos o está cerrada
    if v.Destinox == -1 || v.Destinoy == -1 || a.TiempoEspera >= 60 || a.Estado == "Cerrada" {
        seleccionaAtraccionAlAzar(mapa)
    } else {
        actualizaAtraccion(mapa) // Actualizamos el tiempo de espera de la atracción destino del visitante
    }

    movimiento = calculaMovimiento() // Obtiene el mejor movimiento en base a las posiciones adyacentes y la atracción destino seleccionada
    actualizaPosicion(movimiento) // Actualiza la posición actual del visitante en base al mejor movimiento elegido

    // Si el visitante se encuentra en la atracción
    if (v.Posicionx == v.Destinox) && (v.Posiciony == v.Destinoy) {

        time.Sleep(10 * time.Second) // Esperamos un tiempo para simular el tiempo de ciclo de la atracción

        // Ahora el visitante vuelve a desconocer su destino
        v.Destinox = -1
        v.Destinoy = -1
        a.TiempoEspera = -1
        a.Posicionx = -1
        a.Posiciony = -1
    }
}

return movimiento
}

```

Esta función actuará de forma diferente dependiendo de si el visitante desconoce a qué atracción dirigirse/su atracción actual ha cambiado su tiempo de espera y ahora es mayor o igual a 60 minutos, si dicha atracción ahora está cerrada o si por el contrario no se cumple ninguna de estas dos condiciones.

En el primer caso tenemos que seleccionar una atracción al azar y para ello disponemos de la función `seleccionaAtraccionAlAzar`:

```

/* Función que selecciona una atracción al azar y guarda la posición de dicha atracción en el visitante */
func seleccionaAtraccionALazar(mapa [20][20]string) {

    var atraccionesDisponibles []atraccion

    //Elegimos una atracción al azar del mapa entre las que el tiempo de espera sea menor de 60 minutos
    for i := 0; i < 20; i++ {
        for j := 0; j < 20; j++ {
            // Si la posición actual del mapa es un número, con esto nos basta para que no acuda a una atracción cerrada
            if t, err := strconv.Atoi(mapa[i][j]); err == nil {
                if t < 60 { // Si el tiempo de espera es menor a 60 minutos
                    atraccionAux := atraccion{
                        Posicionx: i,
                        Posiciony: j,
                        TiempoEspera: t,
                        Estado: "Abierta",
                    }
                    atraccionesDisponibles = append(atraccionesDisponibles, atraccionAux)
                }
            }
        }
    }

    // Elegimos al azar una de las atracciones disponibles
    rand.Seed(time.Now().UnixNano()) // Utilizamos la función Seed(semilla) para inicializar la fuente predeterminada al
    min := 0
    max := len(atraccionesDisponibles) - 1
    indexAtraccion := (rand.Intn(max-min+1) + min)

    // Actualizamos las coordenadas de destino del visitante
    v.Destinox = atraccionesDisponibles[indexAtraccion].Posicionx
    v.Destinoy = atraccionesDisponibles[indexAtraccion].Posiciony
    a.Posicionx = atraccionesDisponibles[indexAtraccion].Posicionx
    a.Posiciony = atraccionesDisponibles[indexAtraccion].Posiciony
    a.TiempoEspera = atraccionesDisponibles[indexAtraccion].TiempoEspera
    a.Estado = atraccionesDisponibles[indexAtraccion].Estado

    fmt.Println("Me dirijo a la atracción con tiempo de espera igual a " + strconv.Itoa(a.TiempoEspera))
}

```

Esta se encarga de seleccionar al azar entre las atracciones del mapa que no posean un tiempo de espera mayor o igual a 60 minutos y no estén cerradas, tras lo cual almacena la información de las coordenadas de dicha atracción en la variable global que gestiona el visitante y en la variable global que hace referencia a la atracción destino del visitante, donde además guardamos el tiempo de espera.

Volviendo a la función obtenerMovimiento, en el caso en que no se cumple ninguna de las condiciones mencionadas anteriormente, y por lo tanto, significando que el visitante se dirige a una atracción lo que tenemos que hacer es actualizar el tiempo de espera y el estado de su atracción destino por si hubiese sido actualizado desde el engine tras una petición al servidor de tiempos de espera o por un cambio en el clima de la ciudad asociada al cuadrante en el que se encuentra dicha atracción. Para esto disponemos de la función actualizaAtraccion:

```

/* Función que actualiza el tiempo de espera de la atracción destino del visitante en base al mapa recibido */
func actualizaAtraccion(mapa [20][20]string) {
    tiempoActualizado, err := strconv.Atoi(mapa[a.Posicionx][a.Posiciony])
    if err != nil { // Si se produce un error al convertir a entero quiere decir que la atracción se ha cerrado
        a.Estado = "Cerrada"
    } else {
        a.TiempoEspera = tiempoActualizado // Sino actualizamos el tiempo sin más
    }
}

```

Siguiendo con el código de la función obtenerMovimiento, ahora calculamos el movimiento con la función “calculaMovimiento”:

```
/* Función que devuelve el mejor movimiento a realizar en base a la atracción destino elegida por el visitante */
func calculaMovimiento() string {
    var mejorMovimiento string = ""
    var mejorDistancia int
    var nuevaDistancia int

    xOriginal := v.Posicionx
    yOriginal := v.Posiciony

    // Seleccionamos el mejor movimiento para que el visitante alcance su destino
    for i := 0; i < 8; i++ {

        switch i {
        case 0:
            v.Posicionx--
            if v.Posicionx == -1 {
                v.Posicionx = 19
            }
            mejorDistancia = int(math.Abs(float64(v.Destinox)-float64(v.Posicionx))) + int(math.Abs(float64(v.Destinoy)-float64(v.Posiciony))) // Distancia de Manhattan
            mejorMovimiento = "N"
            v.Posicionx = xOriginal // Reseteamos la posición
        case 1:
            v.Posicionx++
            if v.Posicionx == 20 {
                v.Posicionx = 0
            }
            nuevaDistancia = int(math.Abs(float64(v.Destinox)-float64(v.Posicionx))) + int(math.Abs(float64(v.Destinoy)-float64(v.Posiciony))) // Distancia de Manhattan
            if nuevaDistancia < mejorDistancia {
                mejorDistancia = nuevaDistancia
                mejorMovimiento = "S"
            }
            v.Posicionx = xOriginal // Reseteamos la posición
        case 2:
            v.Posiciony--
            if v.Posiciony == -1 {
                v.Posiciony = 19
            }
            nuevaDistancia = int(math.Abs(float64(v.Destinox)-float64(v.Posicionx))) + int(math.Abs(float64(v.Destinoy)-float64(v.Posiciony))) // Distancia de Manhattan
            if nuevaDistancia < mejorDistancia {
                mejorDistancia = nuevaDistancia
                mejorMovimiento = "W"
            }
            v.Posiciony = yOriginal // Reseteamos la posición
        case 3:
            v.Posiciony++
            if v.Posiciony == 20 {
                v.Posiciony = 0
            }
            nuevaDistancia = int(math.Abs(float64(v.Destinox)-float64(v.Posicionx))) + int(math.Abs(float64(v.Destinoy)-float64(v.Posiciony))) // Distancia de Manhattan
            if nuevaDistancia < mejorDistancia {
                mejorDistancia = nuevaDistancia
                mejorMovimiento = "E"
            }
            v.Posiciony = yOriginal // Reseteamos la posición
        case 4:
            v.Posicionx--
            v.Posiciony--
            if v.Posicionx == -1 {
                v.Posicionx = 19
            }
            if v.Posiciony == -1 {
                v.Posiciony = 19
            }
            nuevaDistancia = int(math.Abs(float64(v.Destinox)-float64(v.Posicionx))) + int(math.Abs(float64(v.Destinoy)-float64(v.Posiciony))) // Distancia de Manhattan
            if nuevaDistancia < mejorDistancia {
                mejorDistancia = nuevaDistancia
                mejorMovimiento = "NW"
            }
            v.Posicionx = xOriginal // Reseteamos la posición
            v.Posiciony = yOriginal // Reseteamos la posición
        case 5:
            v.Posicionx--
            v.Posiciony++
            if v.Posicionx == -1 {
                v.Posicionx = 19
            }
            if v.Posiciony == 20 {
                v.Posiciony = 0
            }
            nuevaDistancia = int(math.Abs(float64(v.Destinox)-float64(v.Posicionx))) + int(math.Abs(float64(v.Destinoy)-float64(v.Posiciony))) // Distancia de Manhattan
            if nuevaDistancia < mejorDistancia {
                mejorDistancia = nuevaDistancia
                mejorMovimiento = "NE"
            }
            v.Posicionx = xOriginal // Reseteamos la posición
            v.Posiciony = yOriginal // Reseteamos la posición
        }
    }
    return mejorMovimiento
}
```

```

        case 6:
            v.Posicionx++
            v.Posiciony--
            if v.Posicionx == 20 {
                v.Posicionx = 0
            }
            if v.Posiciony == -1 {
                v.Posiciony = 19
            }
            nuevaDistancia = int(math.Abs(float64(v.Destinox)-float64(v.Posicionx))) + int(math.Abs(float64(v.Destinoy)-float64(v.Posiciony))) // Distancia de Manhattan
            if nuevaDistancia < mejorDistancia {
                mejorDistancia = nuevaDistancia
                mejorMovimiento = "SW"
            }
            v.Posicionx = xOriginal // Reseteamos la posición
            v.Posiciony = yOriginal // Reseteamos la posición
        case 7:
            v.Posicionx++
            v.Posiciony++
            if v.Posicionx == 20 {
                v.Posicionx = 0
            }
            if v.Posiciony == 20 {
                v.Posiciony = 0
            }
            nuevaDistancia = int(math.Abs(float64(v.Destinox)-float64(v.Posicionx))) + int(math.Abs(float64(v.Destinoy)-float64(v.Posiciony))) // Distancia de Manhattan
            if nuevaDistancia < mejorDistancia {
                mejorDistancia = nuevaDistancia
                mejorMovimiento = "SE"
            }
            v.Posicionx = xOriginal // Reseteamos la posición
            v.Posiciony = yOriginal // Reseteamos la posición
        }
    }
    return mejorMovimiento
}

```

Aquí básicamente lo que hacemos es probar las 8 posibles posiciones y calcular la distancia de Manhattan respecto a cada una de ellas y nos quedaremos con la posición que menor distancia tenga respecto a la atracción destino, y de esta forma devolveremos el mejor movimiento posible en consecuencia.

Volviendo a la función obtenerMovimiento lo siguiente es actualizar la posición del visitante en función del movimiento elegido, y para ello tenemos la función actualizaPosicion:

```

/* Función que actualiza la posición actual del visitante en base al movimiento pasado
func actualizaPosicion(movimiento string) {

    switch movimiento {

        case "N":
            v.Posicionx--
        case "S":
            v.Posicionx++
        case "W":
            v.Posiciony--
        case "E":
            v.Posiciony++
        case "NW":
            v.Posicionx--
            v.Posiciony--
        case "NE":
            v.Posicionx--
            v.Posiciony++
        case "SW":
            v.Posicionx++
            v.Posiciony--
        case "SE":
            v.Posicionx++
            v.Posiciony++
    }

    if v.Posicionx == -1 {
        v.Posicionx = 19
    } else if v.Posicionx == 20 {
        v.Posicionx = 0
    }
}

```

En esta función dependiendo del movimiento elegido actuamos en consecuencia actualizando la posición actual del visitante.

```

// Si el visitante se encuentra en la atracción
if (v.Posicionx == v.Destinox) && (v.Posiciony == v.Destinoy) {

    time.Sleep(10 * time.Second) // Esperamos un tiempo para simular el tiempo de ciclo de la atracción

    // Ahora el visitante vuelve a desconocer su destino
    v.Destinox = -1
    v.Destinoy = -1
    a.TiempoEspera = -1
    a.Posicionx = -1
    a.Posiciony = -1

}

return movimiento

```

Y finalizando con la función “obtenerMovimiento” en caso de que se llegue a la atracción destino hacemos una espera de 10 segundos para simular el tiempo de ciclo de la atracción y reiniciamos los valores de la posición del visitante y de la atracción. Para terminar con la función devolvemos un string con el movimiento elegido que es el que envíamos al engine para que lo procese y nos actualice la posición del visitante actual en el mapa.

```
    movimiento := obtenerMovimiento(mapa)
    peticionMovimiento := v.ID + ":" + movimiento + ":" + strconv.Itoa(v.Destinox) + "," + strconv.Itoa(v.Destinoy)
    peticionMovimientoCifrada, err := encriptacionAES(peticionMovimiento, clave)
    if err != nil {
        panic(err)
    }
    productorMovimientos(IpBroker, PuertoBroker, peticionMovimientoCifrada)

    time.Sleep(1 * time.Second) // Mandamos el movimiento del visitante cada segundo
```

Y volviendo a la función consumidorMapa sólo nos queda mandar la petición con el movimiento al engine para que nos actualice nuestra posición en el mapa, para ello utilizamos la función “productorMovimientos”, habiendo cifrado el mensaje con el movimiento previamente.

Para finalizar con la función consumidorMapa hacemos una espera de 1 segundo para enviar el siguiente movimiento como se nos dice en el enunciado.

Fwq_engine

El engine es el sistema central de la aplicación, el cual pondrá en marcha todo el sistema. Utiliza tres topics, los cuales son los mismos que los que tiene el visitante. Uno para recibir la peticiones por parte de los visitantes, otro para enviar las respuestas de login y otro para enviar el mapa. Además consumirá una API de terceros para obtener la temperatura de 4 ciudades al azar y así determinar si las atracciones de un cuadrante están abiertas o cerradas. Por otro lado la comunicación con el visitante por kafka será cifrada al igual que vimos en el visitante, y cada cierto tiempo guardará en la BD el estado actual del mapa del parque para poder seguir en tiempo real dicho estado desde el front.

Lo primero que nos encontramos en el código del engine es la declaración de la paquetería y la importación de las librerías necesarias:

```
package main

import (
    "bufio"
    "context"
    "crypto/aes"
    "crypto/cipher"
    "database/sql"
    "encoding/base64"
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net"
    "net/http"
    "os"
    "os/signal"
    "runtime/pprof"
    "strconv"
    "strings"
    "time"

    _ "github.com/go-sql-driver/mysql"
    "github.com/segmentio/kafka-go"
    "golang.org/x/crypto/bcrypt"
)
```

Lo segundo que tenemos son las estructuras creadas, las cuales representan a los visitantes, las atracciones y las ciudades. Adicionalmente tenemos un vector al igual que en el visitante que nos va a servir para el cifrado AES de los mensajes intercambiados en el kafka.

```
/**  
 * Estructura del visitante  
 * https://devjaime.medium.com/mi-primer-api-en-golang-9461996753dc  
 */  
You, 2 weeks ago | 2 authors (You and others)  
type visitante struct {  
    ID          string `json:"id"  
    Nombre      string `json:"nombre"  
    Password    string `json:"contraseña"  
    Posicionx   int    `json:"posicionx"  
    Posiciony   int    `json:"posiciony"  
    Destinox    int    `json:"destinox"  
    Destinoy    int    `json:"destinoy"  
    DentroParque int   `json:"dentroParque"  
    IdEnParque  string `json:"idEnParque"  
    UltimoEvento string `json:"ultimoEvento"  
    Parque      string `json:"parqueAtracciones"  
}  
  
/**  
 * Estructura de las atracciones  
 */  
You, 5 days ago | 2 authors (Wilmer Bravo and others)  
type atraccion struct {  
    ID          string `json:"id"  
    TCiclo      int    `json:"tciclo"  
    NVisitantes int   `json:"nvisitantes"  
    Posicionx   int    `json:"posicionx"  
    Posiciony   int    `json:"posiciony"  
    TiempoEspera int   `json:"tiempoEspera"  
    Estado      string `json:"estado"  
    Parque      string `json:"parqueAtracciones"  
}
```

```
You, 3 days ago | 2 authors (You and others)
type ciudad struct {
    Cuadrante string `json:"cuadrante"`
    Nombre     string `json:"name"`
    Temperatura float32 `json:"temp"`
}

// Array de bytes aleatorios para la implementación de seguridad del kafka
var iv = []byte{35, 46, 57, 24, 85, 35, 24, 74, 87, 35, 88, 98, 66, 32, 14, 05}
```

Ahora entramos en el main de la aplicación:

```
func main() {
    IpKafka := os.Args[1]
    PuertoKafka := os.Args[2]
    numeroVisitantes := os.Args[3]
    ciudadesElegidas := os.Args[4]
    IpFWQWaiting := os.Args[5]
    PuertoWaiting := os.Args[6]

    fmt.Println("Creado un engine que atiende peticiones por " + IpKafka + ":" +
        PuertoKafka)

    //Creamos el topic...Cambiar la IpKafka en la función principal
    //Si no se ejecuta el programa, se cierra el kafka?
    crearTopics(IpKafka, PuertoKafka, "peticiones")
    crearTopics(IpKafka, PuertoKafka, "respuesta-login")
    crearTopics(IpKafka, PuertoKafka, "movimiento-mapa")

    // SECURIZAMOS LA COMUNICACIÓN EN KAFKA
    // Cargamos la clave de cifrado AES del archivo
    fichero, err := ioutil.ReadFile("claveCifradoAES.txt")
    if err != nil {
        log.Fatal("Error al leer el archivo de la clave de cifrado AES: ", err)
    }

    clave := string(fichero) // Clave de 32 bits

    // Visitantes, atracciones que se encuentran en la BD
    var visitantesRegistrados []visitante
    var conn = conexionBD()
    maxVisitantes, _ := strconv.Atoi(numeroVisitantes)
    establecerMaxVisitantes(conn, maxVisitantes)
```

Primero nos guardamos los valores introducidos por parámetro, luego nos creamos los 3 topics necesarios, cargamos la clave de cifrado AES y después nos preparamos las variables que van a almacenar a los visitantes registrados, la conexión a la base de datos y el número máximo de visitantes, el cual establecemos además en la BD.

Para obtener la conexión a la base de datos hacemos uso de la siguiente función:

```
/*
 * Función que abre una conexión con la bd
 */
func conexionBD() *sql.DB {
    //Accediendo a la base de datos
    //*****Flate bd*****
    //Abrimos la conexión con la base de datos
    db, err := sql.Open("mysql", "root:1234@tcp(127.0.0.1:3306)/parque_atracciones")
    //Si la conexión falla mostrara este error
    if err != nil {
        panic(err.Error())
    }
    //Cierra la conexión con la bd
    defer db.Close()
    return db
}
```

Y para establecer el máximo número de visitantes en la BD utilizamos esta:

```
/*
 * Función que establece el aforo maximo permitido en el parque de atracciones
 */
func establecerMaxVisitantes(db *sql.DB, numero int) {

    //Ejecutamos la sentencia
    results, err := db.Query("SELECT * FROM parque")

    if err != nil {
        panic("Error al hacer la consulta del parque" + err.Error()) //devolverá nil y error en caso de que no se pueda hacer la consulta
    }

    //Cerramos la base de datos
    defer results.Close()

    //Recorremos los resultados obtenidos por la consulta
    if results.Next() {

        //Variable donde guardamos la información de cada una filas de la sentencia
        sentenciaPreparada, err := db.Prepare("UPDATE parque SET aforoMaximo = ? WHERE id = ?")

        if err != nil {
            panic("Error al preparar la sentencia" + err.Error()) //devolverá nil y error en caso de que no se pueda hacer la consulta
        }

        defer sentenciaPreparada.Close()

        _, err = sentenciaPreparada.Exec(numero, "SDpark")

        if err != nil {
            panic("Error al establecer el número máximo de visitantes" + err.Error())
        }
    }
}
```

```

go consumidorEngine(IPKafka, PuertoKafka, maxVisitantes, clave)

// Guardamos el mapa del parque en curso cada cierto tiempo
go guardarMapaBD()

c := make(chan os.Signal, 1)
signal.Notify(c, os.Interrupt)
go func() {
    for sig := range c {
        log.Printf("captured %v, stopping profiler and exiting..", sig)
        mensaje := "Engine no disponible"
        mensajeCifrado, err := encriptacionAES(string(mensaje), clave)
        if err != nil {
            panic(err)
        }
        mensajeJson, err := json.Marshal(mensajeCifrado)
        if err != nil {
            fmt.Printf("Error a la hora de codificar el mensaje: %v\n", err)
        }

        productorMapa(IPKafka, PuertoKafka, mensajeJson)
    }
}

func main() {
    fmt.Println()
    fmt.Println("Engine apagado manualmente")
    pprof.StopCPUProfile()
    os.Exit(1)
}

```

Ahora tenemos la llamada a 3 go routines que nos van a servir para procesar las peticiones realizadas por parte de los visitantes, guardar el mapa cada segundo en la BD y controlar la caída del engine y enviar una notificación a los visitantes que se encuentran en el parque. Para gestionar esta situación hemos hecho uso de los channels de golang y simplemente comentar que la notificación la enviamos por medio de la función ya vista, productorMapa.

Entrando en la función “consumidorEngine”, lo primero que nos encontramos en la definición/configuración del consumidor de kafka junto con una nueva apertura de conexión a la base de datos que no dependa de la que trabaja en el hilo de ejecución principal del main:

```
/* Función que recibe del gestor de colas las credenciales de los visitantes que quieren iniciar sesión para entrar en el parque */
func consumidorEngine(IpKafka, PuertoKafka string, ctx context.Context, maxVisitantes int) {

    //Accediendo a la base de datos
    //Abrimos la conexión con la base de datos
    db, err := sql.Open("mysql", "root:1234@tcp(127.0.0.1:3306)/parque_atracciones")
    //Si la conexión falla mostrara este error
    if err != nil {
        panic(err.Error())
    }
    //Cierra la conexión con la bd
    defer db.Close()

    direcciónKafka := IpKafka + ":" + PuertoKafka
    //Configuración de lector de kafka
    conf := kafka.ReaderConfig{
        Brokers:      []string{direcciónKafka},
        Topic:        "peticiones", //Topico que hemos creado
        GroupID:     "visitantes",
        StartOffset: kafka.LastOffset,
    }

    reader := kafka.NewReader(conf)
```

Continuando con el código de la función aparece el bucle for donde iremos leyendo los mensajes que han sido encolados por los visitante en el gestor de colas:

```

for {
    m, err := reader.ReadMessage(context.Background())
    if err != nil {
        fmt.Println("Ha ocurrido algún error a la hora de conectarse con el kafka", err)
    }
    cadena, err := desencriptacionAES(string(m.Value), clave)
    if err != nil {
        panic(err)
    }
    fmt.Println("Petición recibida: " + cadena)
    cadenaPeticion := strings.Split(cadena, ":")

    alias := cadenaPeticion[0]
    peticion := cadenaPeticion[1]
    destino := strings.Split(cadenaPeticion[2], ",")
    destinoX, _ := strconv.Atoi(strings.TrimSpace(destino[0]))
    destinoY, _ := strconv.Atoi(strings.TrimSpace(destino[1]))

    v := visitante{
        ID:      strings.TrimSpace(alias),
        Password: strings.TrimSpace(peticion),
        Destinox: destinoX,
        Destinoy: destinoY,
    }

    // Nos guardamos la posible contraseña recibida
    var contraseña string = v.Password

    // Obtenemos el hash de la contraseña del visitante en caso de que el ID coincida con alguno almacenado en la BD
    results := db.QueryRow("SELECT contraseña FROM visitante WHERE id = ?", v.ID)

    var hash string
    results.Scan(&hash)

    var respuesta string = ""

```

En este bucle procesamos los mensajes recibidos, los desencriptamos con la misma función antes vista en el visitante y obtenemos el alias del visitante, la petición realizada, y las coordenadas de destino actuales de dicho visitante. A continuación nos creamos una instancia del struct visitante para gestionar/almacenar dicha información del visitante y tras esto hacemos una consulta a la base de datos que nos va a permitir comprobar si lo recibido son credenciales de acceso, en cuyo caso se trata de una petición de login para entrar en el parque. Para terminar con esta captura nos vamos ya preparando la variable respuesta donde almacenaremos la respuesta a la petición de login.

Ahora entramos en las diferentes casuísticas que se pueden presentar en esta función.

```

// Si las credenciales coinciden con las de un visitante registrado en la BD y el parque no está lleno
if CheckPasswordHash(contraseña, hash) && !parqueLleno(db, maxVisitantes) {

    // Actualizamos el estado del visitante en la BD
    sentenciaPreparada, err := db.Prepare("UPDATE visitante SET dentroParque = 1, destinox = ?, destinoy = ? WHERE id = ?")
    if err != nil {
        panic("Error al preparar la sentencia de modificación: " + err.Error())
    }

    // Ejecutar sentencia, un valor por cada '?'
    _, err = sentenciaPreparada.Exec(v.Destinox, v.Destinoy, v.ID)
    if err != nil {
        panic("Error al actualizar el estado del visitante respecto al parque: " + err.Error())
    }

    // Nos guardamos los visitantes del parque asociados a este engine
    //visitantesDelEngine = append(visitantesDelEngine, v.ID)

    respuesta += alias + ":" + "Acceso concedido"
    respuestaCifrada, err := encriptacionAES(respuesta, clave)
    if err != nil {
        panic(err)
    }
    productorLogin(IpKafka, PuertoKafka, respuestaCifrada)

    sentenciaPreparada.Close()
}

```

La primera de ellas se trata del caso en el que hemos recibido una petición de login, por lo que comprobamos si las credenciales coinciden con las de un visitante registrado en la BD y que el parque no esté lleno actualmente. Para la comprobación de las credenciales hemos usado esta función auxiliar para comprobar si el hash del password recibido coincide con el almacenado en la BD:

```

/* Función que nos sirve para comprobar el hash de un password */
func CheckPasswordHash(password, hash string) bool {
    err := bcrypt.CompareHashAndPassword([]byte(hash), []byte(password))
    return err == nil
}

```

Y para la comprobación de si el parque está lleno hemos usado esta:

```

/* Función que comprueba si el aforo del parque está completo o no */
func parqueLleno(db *sql.DB, maxAforo int) bool {
    var lleno bool = false

    // Comprobamos si las credenciales de acceso son válidas
    results, err := db.Query("SELECT * FROM visitante WHERE dentroParque = 1")

    if err != nil {
        fmt.Println("Error al hacer la consulta sobre la BD para comprobar el aforo: " + err.Error())
    }

    visitantesDentroParque := 0 // Variable en la que vamos a almacenar el número de visitantes que se encuentran en el parque

    // Vamos recorriendo las filas devueltas para obtener el nº de visitantes dentro del parque
    for results.Next() {
        visitantesDentroParque++
    }

    results.Close() // cerramos la conexión a la BD

    // Si el aforo está al completo
    if visitantesDentroParque >= maxAforo {
        lleno = true
    }

    return lleno
}

```

You, 7 months ago • Login de visitantes implementado a falta de contr...

En caso de que dichas condiciones se cumplan, actualizamos el estado del visitante en el BD, estableciendo su atracción destino actual, la cual será desconocida (valores -1 y -1) e indicando que ahora se encuentra dentro del parque.

Si todo va bien formamos el mensaje de respuesta donde le indicamos al visitante que le permitimos el acceso al parque y lo envíamos utilizando para ello la función “productorLogin”:

```

/* Función que envía el mensaje de respuesta a la petición de login de un visitante */
func productorLogin(IpBroker, PuertoBroker string, ctx context.Context, respuesta string) {

    var brokerAddress string = IpBroker + ":" + PuertoBroker
    var topic string = "respuesta-login"

    w := kafka.NewWriter(kafka.WriterConfig{
        Brokers:      []string{brokerAddress},
        Topic:        topic,
        CompressionCodec: kafka.Snappy.Codec(),
    })

    err := w.WriteMessages(ctx, kafka.Message{
        Key:   []byte("Key-Login"),
        Value: []byte(respuesta),
    })

    if err != nil {
        fmt.Println("No se puede mandar el mensaje de respuesta a la petición de login: " + err.Error())
    }
}

```

La segunda de las casuísticas que se pueden dar es la referente a una solicitud de movimiento:

```

else if peticion == "IN" || peticion == "N" || peticion == "S" || peticion == "W" || peticion == "E" || peticion == "NW" ||
peticion == "NE" || peticion == "SW" || peticion == "SE" { // Si se nos ha mandado un movimiento

    // Comprobamos que el alias pertenezca a un visitante que se encuentra en el parque
    results, err := db.Query("SELECT * FROM visitante WHERE id = ?", v.ID)

    if err != nil {
        fmt.Println("Error al hacer la consulta sobre la BD para el login: " + err.Error())
    }

    // Actualizamos el estado el destino del visitante en la BD
    sentenciaPreparada, err := db.Prepare("UPDATE visitante SET destinox = ?, destinoy = ? WHERE id = ?")
    if err != nil {
        panic("Error al preparar la sentencia de modificación de destino: " + err.Error())
    }

    // Ejecutar sentencia, un valor por cada '?'
    _, err = sentenciaPreparada.Exec(v.Destinox, v.Destinoy, v.ID)
    if err != nil {
        panic("Error al actualizar el destino del visitante: " + err.Error())
    }

    sentenciaPreparada.Close()
}

```

En este caso tenemos varios posibles movimientos a recibir, pero si hay uno que quiero destacar es el “IN” que hace referencia al movimiento ejercido por el visitante para indicarnos que ha entrado en el parque. El resto de movimientos son los que se mencionan en el enunciado y que se comportan tal y como indican sus siglas.

Entrando en la lógica de este bloque de código, primero hacemos una select para luego comprobar que el alias indicado en la petición de movimiento pertenezca a un visitante que se encuentra en el parque y luego actualizamos el destino del visitante en la BD.

```

if results.Next() {

    var mapa [20][20]string
    visitantesParque, _ := obtenerVisitantesParque(db)           // Obtenemos los visitantes del parque actualizados
    mueveVisitante(db, alias, peticion, visitantesParque)       // Movemos al visitante en base al movimiento recibido
    visitantesParqueActualizados, _ := obtenerVisitantesParque(db) // Obtenemos los visitantes del parque actualizados
    // Preparamos el mapa a enviar a los visitantes que se encuentran en el parque
    atracciones, _ := obtenerAtraccionesBD(db) // Obtenemos las atracciones actualizadas
    mapaActualizado := asignacionPosiciones(visitantesParqueActualizados, atracciones, mapa)
    var representacion string
    for i := 0; i < len(mapaActualizado); i++ {
        for j := 0; j < len(mapaActualizado); j++ {
            if j == 19 {
                representacion = representacion + mapaActualizado[i][j] + "\n"
            } else {
                representacion = representacion + mapaActualizado[i][j]
            }
        }
    }

    // Encriptamos el mapa
    representacionCifrada, err := encriptacionAES(representacion, clave)
    if err != nil {
        panic(err)
    }

    //Convertimos el mapaActualizado a formato json
    //Esta función devuelve un array de byte
    mapaJson, err := json.Marshal(representacionCifrada)
    //En formato json tiene cuenta el salto de linea por lo que hay que ver si al decodificarlo se quita
    if err != nil {
        fmt.Println("Error a la hora de codificar el mapa: %v\n", err)
    }

    productorMapa(IpKafka, PuertoKafka, mapaJson) // Mandamos el mapa actualizado a los visitantes que se encuentran en el parque
    results.Close()
}

```

En caso que el id de la petición sea válido vamos a formar el mapa.

Para ello empezamos obteniendo los visitantes que se encuentran actualmente en el parque, utilizando para ello la función “obtenerVisitantesParque”:

```
/*
 * Función que obtiene todos los visitantes que se encuentran en el parque de la BD
 * @return []visitante : Arrays de los visitantes obtenidos en la sentencia
 * @return error : Error en caso de que no se haya podido obtener ninguno
 */
func obtenerVisitantesParque(db *sql.DB) ([]visitante, error) {
    //Ejecutamos la sentencia
    results, err := db.Query("SELECT * FROM visitante WHERE dentroParque = 1")

    if err != nil {
        return nil, err //devolvera nil y error en caso de que no se pueda hacer la consulta
    }

    //Cerramos la base de datos
    defer results.Close()

    //Declaramos el array de visitantes
    var visitantes []visitante

    //Recorremos los resultados obtenidos por la consulta
    for results.Next() {
        //Variable donde guardamos la información de cada una filas de la sentencia
        var fwq_visitante visitante

        if err := results.Scan(&fwq_visitante.ID, &fwq_visitante.Nombre,
            &fwq_visitante.Password, &fwq_visitante.Posicionx,
            &fwq_visitante.Posiciony, &fwq_visitante.Destinox, &fwq_visitante.Destinoy,
            &fwq_visitante.DentroParque, &fwq_visitante.IdEnParque, &fwq_visitante.UltimoEvento, &fwq_visitante.Parque); err != nil {
            return visitantes, err
        }

        //Vamos añadiendo los visitantes al array
        visitantes = append(visitantes, fwq_visitante)
    }

    if err = results.Err(); err != nil {
        return visitantes, err
    }

    return visitantes, nil
}
```

Ahora procesamos el movimiento del visitante para poder actualizar el mapa con la función “mueveVisitante”:

```

/* Función que modifica las posiciones de los visitantes en el parque en base a sus movimientos */
func mueveVisitante(db *sql.DB, id, movimiento string, visitantes []visitante) {

    var nuevaPosicionX int
    var nuevaPosicionY int

    for i := 0; i < len(visitantes); i++ {

        if id == visitantes[i].ID { // Modificamos la posición del visitante recibido por kafka

            switch movimiento {
            case "N":
                visitantes[i].Posicionx--
            case "S":
                visitantes[i].Posicionx++
            case "W":
                visitantes[i].Posiciony--
            case "E":
                visitantes[i].Posiciony++
            case "NW":
                visitantes[i].Posicionx--
                visitantes[i].Posiciony--
            case "NE":
                visitantes[i].Posicionx--
                visitantes[i].Posiciony++
            case "SW":
                visitantes[i].Posicionx++
                visitantes[i].Posiciony--
            case "SE":
                visitantes[i].Posicionx++
                visitantes[i].Posiciony++
            }

            if visitantes[i].Posicionx == -1 {
                visitantes[i].Posicionx = 19
            } else if visitantes[i].Posicionx == 20 {
                visitantes[i].Posicionx = 0
            }

            if visitantes[i].Posiciony == -1 {
                visitantes[i].Posiciony = 19
            } else if visitantes[i].Posiciony == 20 {
                visitantes[i].Posiciony = 0
            }

            nuevaPosicionX = visitantes[i].Posicionx
            nuevaPosicionY = visitantes[i].Posiciony
        }
    }

    // MODIFICAMOS la posición de dicho visitante en la BD
    // Preparamos para prevenir inyecciones SQL
    sentenciaPreparada, err := db.Prepare("UPDATE visitante SET posicionx = ?, posiciony = ? WHERE id = ?")
    if err != nil {
        panic("Error al preparar la sentencia de modificación: " + err.Error())
    }

    defer sentenciaPreparada.Close()

    // Ejecutar sentencia, un valor por cada '?'
    _, err = sentenciaPreparada.Exec(nuevaPosicionX, nuevaPosicionY, id)
    if err != nil {
        panic("Error al modificar la posición del visitante en la BD: " + err.Error())
    }
}

```

En esta función dependiendo del movimiento recibido actualizaremos las coordenadas del visitante en el mapa en consecuencia, teniendo en cuenta que si se alcanza un valor límite y al ser un mapa circular tenemos que actualizar la coordenada por la del extremo contrario.

Tras esto y volviendo a la segunda casuística, volvemos a obtener los visitantes del parque actualizados, obtenemos además las atracciones actualizadas y ahora generemos/actualizamos el mapa con la función “asignacionPosiciones”:

```

/*
 * Función que forma el mapa del parque conteniendo a los visitantes y las atracciones
 * @return [20][20]string : Matriz bidimensional representando el mapa
 */
func asignacionPosiciones(visitantes []visitante, atracciones []atraccion, mapa [20][20]string) [20][20]string {
    //Asignamos los id de los visitantes
    for i := 0; i < len(mapa); i++ {
        for j := 0; j < len(mapa[i]); j++ {
            for k := 0; k < len(visitantes); k++ {
                if i == visitantes[k].Posicionx && j == visitantes[k].Posiciony && visitantes[k].DentroParque == 1 {
                    mapa[i][j] = visitantes[k].IdEnParque + "|"
                }
            }
        }
    }

    //Asignamos los valores de tiempo de espera de las atracciones
    for i := 0; i < len(mapa); i++ {
        for j := 0; j < len(mapa[i]); j++ {
            for k := 0; k < len(atracciones); k++ {
                if i == atracciones[k].Posicionx && j == atracciones[k].Posiciony && atracciones[k].Estado == "Abierta" {
                    mapa[i][j] = strconv.Itoa(atracciones[k].TiempoEspera) + "|"
                } else if i == atracciones[k].Posicionx && j == atracciones[k].Posiciony && atracciones[k].Estado == "Cerrada" {
                    mapa[i][j] = "(" + strconv.Itoa(atracciones[k].TiempoEspera) + ")" + "|"
                }
            }
        }
    }

    // Las casillas del mapa que no tengan ni visitantes ni atracciones las representamos con una guión
    for i := 0; i < len(mapa); i++ {
        for j := 0; j < len(mapa[i]); j++ {
            if len(mapa[i][j]) == 0 {
                mapa[i][j] = "-" + "|"
            }
        }
    }
    return mapa
}

```

Un detalle a tener en cuenta es que para representar las atracciones que están cerradas, estas en el mapa aparecerán con un paréntesis a su alrededor.

Tras obtener el mapa devuelto por esta función lo procesamos con un par de bucles for para generar los espacios correspondientes a las filas y después lo encriptamos y convertimos a formato json para enviarlo al visitante utilizando para ello la función “productorMapa”:

```

/* Función que envia el mapa a los visitantes */
func productorMapa(IpBroker, PuertoBroker string, ctx context.Context, mapa []byte) {

    var brokerAddress string = IpBroker + ":" + PuertoBroker
    var topic string = "movimiento-mapa"

    w := kafka.NewWriter(kafka.WriterConfig{
        Brokers:          []string{brokerAddress},
        Topic:            topic,
        CompressionCodec: kafka.Snappy.Codec(),
    })

    err := w.WriteMessages(ctx, kafka.Message{
        Key:   []byte("Key-Mapa"), //[]byte(strconv.Itoa(i)),
        Value: []byte(mapa),
    })
    if err != nil {
        panic("No se puede mandar el mapa: " + err.Error())
    }
}

```

Por el contrario, si el alias indicado en la petición de movimiento no es válido entonces:

```

} else { // Si el alias no pertenece a un visitante del parque
    respuesta += alias + ":" + "Parque cerrado"
    productorLogin(IpKafka, PuertoKafka, ctx, respuesta)
    results.Close()
}

```

Notificamos al engine utilizando la función de productorLogin vista anteriormente, indicando que el parque está cerrado.

```

    // Si se nos ha solicitado una salida del parque
} else if petición == "OUT" {

    // Sacamos del parque al visitante y reiniciamos tanto su posición actual como su destino
    sentenciaPreparada, err := db.Prepare("UPDATE visitante SET dentroParque = 0, posicionx = 0, posiciony = 0, destinox = -1, destinoy = -1 WHERE id = ?")
    if err != nil {
        panic("Error al preparar la sentencia de modificación: " + err.Error())
    }

    // Ejecutar sentencia, un valor por cada '?'
    _, err = sentenciaPreparada.Exec(v.ID)
    if err != nil {
        panic("Error al actualizar el estado del visitante respecto al parque: " + err.Error())
    }

    sentenciaPreparada.Close()

    RegistroLog(db, IpKafka+":"+PuertoKafka, v.ID, "Baja", "El visitante "+v.ID+" ha salido del parque") // Registrarmos el evento de log
}

```

La tercera casuística de la función “consumidorEngine” es aquella en la que se recibe una petición de salida del parque por parte de un visitante, para lo cual reiniciamos la información de dicho visitante en la BD, y registramos el evento de log en la tabla del visitante.

```

} else { // Si las credenciales enviadas para iniciar sesión no son válidas

    if parqueLleno(db, maxVisitantes) {
        respuesta += alias + ":" + "Aforo al completo"
        respuestaCifrada, err := encriptacionAES(respuesta, clave)
        if err != nil {
            panic(err)
        }
        productorLogin(IpKafka, PuertoKafka, respuestaCifrada)
    } else {
        respuesta += alias + ":" + "Parque cerrado"
        respuestaCifrada, err := encriptacionAES(respuesta, clave)
        if err != nil {
            panic(err)
        }
        productorLogin(IpKafka, PuertoKafka, respuestaCifrada)
    }
}

```

Y la cuarta y última casuística de la función “consumidorEngine” es aquella en la que las credenciales de login no son válidas o bien el aforo está completo, enviando al visitante un mensaje cifrado informando del error/situación con la función “productorLogin” ya vista.

```

for {
    visitantesRegistrados, _ = obtenerVisitantesBD(conn) // Obtenemos los visitantes registrados actualmente
    fmt.Println("***** FUN WITH QUEUES RESORT ACTIVITY MAP *****")
    fmt.Println("ID      " + "     Nombre      " + " Pos.      " + " Destino      " + " DentroParque")
    //Hay que usar la función TrimSpace porque al parecer tras la obtención de valores de BD se agrega un retorno de carro a cada variable
    //Mostramos los visitantes registrados en la aplicación actualmente
    for i := 0; i < len(visitantesRegistrados); i++ {
        fmt.Println(strings.TrimSpace(visitantesRegistrados[i].ID) + "      " + strings.TrimSpace(visitantesRegistrados[i].Nombre) +
                   "      " + "(" + strings.TrimSpace(strconv.Itoa(visitantesRegistrados[i].Posicionx)) + "," + strings.TrimSpace(strconv.Itoa(visitantesRegistrados[i].Destinox)) + ")" + "      " + strings.TrimSpace(strconv.Itoa(visitantesRegistrados[i].Dentroparque)))
    }

    fmt.Println() // Para mejorar la visualización

    // Cada X segundos se conectará al servidor de tiempos para actualizar los tiempos de espera de las atracciones
    time.Sleep(time.Duration(5 * time.Second))
    atracciones, _ := obtenerAtraccionesBD(conn) // Obtenemos las atracciones actualizadas
    conexionTiempoEspera(conn, IpFWWaiting, PuertoWaiting, atracciones)

    actualizarClimaParque(ciudadesElegidas, atracciones)

    fmt.Println() // Para mejorar la visualización
}

```

Para terminar con el main tenemos un bucle for indefinido en el que iremos mostrando de forma actualizada el estado de los visitantes que se encuentran registrados en la aplicación/el parque y tras esto hacemos una espera de 5 segundos para después realizar una petición al servidor de tiempos en la que solicitamos que se nos envíen los tiempos de espera de las atracciones del parque actualizados. Esto lo hacemos con la función `conexionTiempoEspera`:

```

/*
 * Función que se conecta al servidor de tiempos para obtener los tiempos de espera actualizados
 */
func ConexionTiempoEspera(db *sql.DB, IpFWWaiting, PuertoWaiting string, atracciones []atraccion) {
    fmt.Println() // Por limpieza
    fmt.Println("****Conexión con el servidor de tiempo de espera****")
    fmt.Println() // Por limpieza
    //fmt.Println("Arrancando el engine para atender los tiempos en el puerto: " + IpFWWaiting + ":" + PuertoWaiting)
    var connType string = "tcp"
    conn, err := net.Dial(connType, IpFWWaiting+":"+PuertoWaiting)

    if err != nil {
        log.Println("ERROR: El servidor de tiempos de espera no está disponible", err.Error())
    } else {

        fmt.Println("****Actualizando los tiempos de espera****")
        fmt.Println() // Por limpieza

        var infoAtracciones string = ""

        for i := 0; i < len(atracciones); i++ {
            infoAtracciones += atracciones[i].ID + ":"
            infoAtracciones += strconv.Itoa(atracciones[i].TCiclo) + ":"
            infoAtracciones += strconv.Itoa(atracciones[i].NVisitantes) + ":"
            infoAtracciones += strconv.Itoa(atracciones[i].TiempoEspera) + "|"
        }

        infoAtracciones += "\n" // Le añadimos el salto de línea porque los sockets los estamos leyendo hasta final de línea
        fmt.Println("Enviendo información de las atracciones...")

        conn.Write([]byte(infoAtracciones)) // Mandamos el id:tiempoCiclo:nºvisitantes de cada atracción en un string
        tiemposEspera, _ := bufio.NewReader(conn).ReadString('\n') // Obtenemos los tiempos de espera actualizados

        if tiemposEspera != "" {
            log.Println("Tiempos de espera actualizados: " + tiemposEspera)
            arrayTiemposEspera := strings.Split(tiemposEspera, "|")

            // Actualizamos los tiempos de espera de las atracciones en la BD
            actualizaTiempoEsperaBD(db, arrayTiemposEspera)
        } else {
            log.Println("Servidor de tiempos no disponible.")
        }
    }
}

```

En esta función nos conectamos vía socket con el servidor de tiempos de espera y dependiendo de si este está disponible o no actuaremos en consecuencia.

Si no está disponible entonces simplemente mostramos un error por pantalla indicando dicha situación sin hacer nada más. Por el contrario, si se encuentra disponible, entonces mandamos la petición vía socket conteniendo toda la información de las atracciones actual y leemos la respuesta del servidor, con la cual, actualizamos los tiempos de espera de la atracciones en la base de datos, utilizando para ello la función `actualizaTiempoEsperaBD`:

```

/* Función que actualiza los tiempos de espera de las atracciones en la BD */
func actualizaTiemposEsperaBD(db *sql.DB, tiemposEspera []string) {
    results, err := db.Query("SELECT * FROM atraccion")

    // Comprobamos que no se produzcan errores al hacer la consulta
    if err != nil {
        panic("Error al hacer la consulta a la BD: " + err.Error())
    }

    defer results.Close() // Nos aseguramos de cerrar*/

    i := 0

    // Recorremos todas las filas de la consulta
    for results.Next() {

        // Preparamos para prevenir inyecciones SQL
        sentenciaPreparada, err := db.Prepare("UPDATE atraccion SET tiempoEspera = ? WHERE id = ?")
        if err != nil {
            panic("Error al preparar la sentencia de modificación: " + err.Error())
        }

        defer sentenciaPreparada.Close()

        infoAtraccion := strings.Split(tiemposEspera[i], ":") // Extraemos el id y el tiempo de espera de la atracción
        idAtraccion := infoAtraccion[0]

        nuevoTiempo, err := strconv.Atoi(infoAtraccion[1])

        if err != nil {
            panic("Error al convertir la cadena con el nuevo tiempo de la atracción")
        }

        // Ejecutar sentencia, un valor por cada ?'
        _, err = sentenciaPreparada.Exec(nuevoTiempo, idAtraccion)
        if err != nil {
            panic("Error al modificar el tiempo de espera de la atracción: " + err.Error())
        }

        i++
    }
}

```

Entonces si todo ha ido bien, se mostrará un mensaje por pantalla informando al respecto y en caso contrario lo mismo.

Volviendo al bucle for principal del main, tenemos una última llamada a una nueva función llamada “actualizarClimaParque”:

```
/* Función que muestra un menú para poder seleccionar las 4 ciudades del listado */
func actualizarClimaParque(numerosCiudadesElegidas string, atracciones []atraccion) {

    // Cargamos la clave de cifrado AES del archivo
    ficheroCiudades, err := ioutil.ReadFile("ciudades.txt")
    if err != nil {
        log.Fatal("No se ha podido leer las ciudades del archivo txt")
    }

    nombresCiudades := strings.Split(string(ficheroCiudades), ",")
    ciudadesElegidas := seleccionaCiudades(nombresCiudades, numerosCiudadesElegidas)
    var ciudades []ciudad

    for i, nombreCiudad := range ciudadesElegidas {    You, now * Uncommitted changes
        city := ciudad{}

        switch i {
        case 0:
            city.Cuadrante = "arriba-izquierda"
        case 1:
            city.Cuadrante = "arriba-derecha"
        case 2:
            city.Cuadrante = "abajo-izquierda"
        case 3:
            city.Cuadrante = "abajo-derecha"
        }

        city.Nombre = nombreCiudad
        city.Temperatura = obtenerClimaCiudad(nombreCiudad)

        ciudades = append(ciudades, city)
    }

    fmt.Println()
    fmt.Println("La ciudad del cuadrante arriba-izquierda es:", ciudades[0].Nombre, "y su temperatura es: ", ciudades[0].Temperatura, "°C")
    fmt.Println("La ciudad del cuadrante arriba-derecha es:", ciudades[1].Nombre, "y su temperatura es: ", ciudades[1].Temperatura, "°C")
    fmt.Println("La ciudad del cuadrante abajo-izquierda es:", ciudades[2].Nombre, "y su temperatura es: ", ciudades[2].Temperatura, "°C")
    fmt.Println("La ciudad del cuadrante abajo-derecha es:", ciudades[3].Nombre, "y su temperatura es: ", ciudades[3].Temperatura, "°C")
    fmt.Println()

    almacenarCiudades(ciudades) // Almacenamos las ciudades en la BD para poder consultar su información desde el front

    // Actualizamos la situación del parque en base a las temperaturas de las ciudades
    // OTRA OPCIÓN ES CREAR UNA VARIABLE GLOBAL CON EL ESTADO ACTUAL DE LAS 4 CIUDADES
    actualizarClimaAtracciones(ciudades, atracciones)
}
```

Esta función se encarga de actualizar el clima del parque, y por lo tanto cerrar las correspondientes atracciones, esto es, dependiendo de la temperatura obtenida de las 4 ciudades.

Para ello, lo primero que hacemos es obtener el nombre de las 4 ciudades del fichero de texto, y utilizando la función “seleccionaCiudades” nos guardamos en un array el nombre de dichas ciudades:

```
/* Función que selecciona las 4 ciudades elegidas */
func seleccionaCiudades(ciudades []string, numCiudadesElegidas string) []string {
    numCiudades := strings.Split(numCiudadesElegidas, ",")
    var nombresCiudades []string
    for i := 0; i < 4; i++ {
        num, _ := strconv.Atoi(numCiudades[i])
        nombresCiudades = append(nombresCiudades, ciudades[num-1])
    }
    return nombresCiudades
}
```

Esta función en base al parámetro del engine donde indicamos los 4 dígitos de las ciudades que queremos del fichero almacena los nombre en un array y lo devuelve.

Siguiendo con la función “actualizarClimaParque”, asociamos los cuadrantes a cada ciudad y luego con la función “obtenerClimaCiudad” nos guardamos la temperatura obtenida de cada ciudad.

Para después mostrar toda la información por consola y almacenar a continuación dicha información en la BD para después poder servirla al front desde el api engine.

Respecto a la función “obtenerClimaCiudad”:

```
func obtenerClimaCiudad(nombreCiudad string) float32 {
    clienteHttp := &http.Client{}

    // Cargamos elapikey desde fichero para consumir la api de OpenWeather
    fichero, err := ioutil.ReadFile("apikey.txt")
    if err != nil {
        log.Fatalf("Error al leer el archivo de la api key de OpenWeather: ", err)
    }
    var apiKey string = string(fichero)

    peticion, err := http.NewRequest("GET", "https://api.openweathermap.org/data/2.5/weather?q="+nombreCiudad+"&appid="+apiKey+"&lang=es"&units=metric", nil)

    if err != nil {
        log.Fatalf("Error creando petición: %v", err)
    }

    peticion.Header.Add("Content-Type", "application/json")

    respuesta, err := clienteHttp.Do(peticion)
    if err != nil {
        // Maneja el error de acuerdo a tu situación
        log.Fatalf("Error haciendo petición: %v", err)
    }

    // No olvides cerrar el cuerpo al terminar
    defer respuesta.Body.Close()

    // Vamos a obtener el cuerpo y lo almacenaremos en la variable
    cuerpoRespuesta, err := ioutil.ReadAll(respuesta.Body)
    //var city ciudad
    //body := json.NewDecoder(peticion.Body).Decode(&city)
    if err != nil {
        log.Fatalf("Error leyendo respuesta: %v", err)
    }
    respuestaString := strings.Split(string(cuerpoRespuesta), ",")
    respuestaString = strings.Split(respuestaString[7], ";")
    //fmt.Println(respuestaString[2])

    // Convertimos el string a float32
    value, err := strconv.ParseFloat(respuestaString[2], 32)
    if err != nil {
        panic(err)
    }
    temperatura := float32(value)

    return temperatura
}
```

Primero cargamos elapikey para poder consumir la api de terceros desde fichero, y usamos dichaapikey para preparar la petición GET junto con el nombre de la ciudad en cada caso.

Añadimos la cabecera content-type, indicando el contenido de la petición en json, y realizamos la petición mencionada con Do, para después obtener el body de respuesta, convertirlo a string para procesarlo y extraer la temperatura de la ciudad solicitada, la cual convertimos de string a float32 para su posterior devolución.

Volviendo a la función “actualizarClimaParque”, esta finaliza con la llamada a la función “actualizarClimaAtracciones”, la cual actualiza el estado de las atracciones en base al clima del cuadrante donde se encuentre:

```
/* Función que actualiza el estado de las atracciones en base al clima del cuadrante donde se encuentre */
func actualizarClimaAtracciones(ciudades []ciudad, atracciones []atraccion) {

    //Accediendo a la base de datos
    //Abrimos la conexión con la base de datos
    db, err := sql.Open("mysql", "root:1234@tcp(127.0.0.1:3306)/parque_atracciones")
    //Si la conexión falla mostrara este error
    if err != nil {
        panic(err.Error())
    }
    //Cierra la conexión con la bd
    defer db.Close()

    for _, atraccion := range atracciones {

        // Si la atraccion se encuentra en el cuadrante arriba-izquierda
        if atraccion.Posicionx >= 0 && atraccion.Posicionx <= 9 && atraccion.Posiciony >= 0 && atraccion.Posiciony <= 9 {

            // Comprobamos el clima de la ciudad asociada
            if ciudades[0].Temperatura < 20.0 || ciudades[0].Temperatura > 30.0 {

                // Actualizamos el estado de la atracción en la BD
                sentenciaPreparada, err := db.Prepare("UPDATE atraccion SET estado = ? WHERE id = ?")
                if err != nil {
                    panic("Error al preparar la sentencia de modificación del estado de la atracción: " + err.Error())
                }

                defer sentenciaPreparada.Close()

                // Ejecutar sentencia, un valor por cada '?'
                _, err = sentenciaPreparada.Exec("Cerrada", atraccion.ID)
                if err != nil {
                    panic("Error al actualizar el estado de la atracción: " + err.Error())
                }

            } else {
                // Actualizamos el estado de la atracción en la BD
                sentenciaPreparada, err := db.Prepare("UPDATE atraccion SET estado = ? WHERE id = ?")
                if err != nil {
                    panic("Error al preparar la sentencia de modificación del estado de la atracción: " + err.Error())
                }

                defer sentenciaPreparada.Close()

                // Ejecutar sentencia, un valor por cada '?'
                _, err = sentenciaPreparada.Exec("Abierta", atraccion.ID)
                if err != nil {
                    panic("Error al actualizar el estado de la atracción: " + err.Error())
                }
            }
        }
    }
}
```

```

    // Si la atraccion se encuentra en el cuadrante arriba-derecha
} else if atraccion.Posicionx >= 0 && atraccion.Posicionx <= 9 && atraccion.Posiciony >= 10 && atraccion.Posiciony <= 19 {

    // Comprobamos el clima de la ciudad asociada
    if ciudades[1].Temperatura < 20.0 || ciudades[1].Temperatura > 30.0 {

        // Actualizamos el estado de la atracción en la BD
        sentenciaPreparada, err := db.Prepare("UPDATE atraccion SET estado = ? WHERE id = ?")
        if err != nil {
            panic("Error al preparar la sentencia de modificación del estado de la atracción: " + err.Error())
        }

        defer sentenciaPreparada.Close()

        // Ejecutar sentencia, un valor por cada '?'
        _, err = sentenciaPreparada.Exec("Cerrada", atraccion.ID)
        if err != nil {
            panic("Error al actualizar el estado de la atracción: " + err.Error())
        }
    }

} else {
    // Actualizamos el estado de la atracción en la BD
    sentenciaPreparada, err := db.Prepare("UPDATE atraccion SET estado = ? WHERE id = ?")
    if err != nil {
        panic("Error al preparar la sentencia de modificación del estado de la atracción: " + err.Error())
    }

    defer sentenciaPreparada.Close()

    // Ejecutar sentencia, un valor por cada '?'
    _, err = sentenciaPreparada.Exec("Abierta", atraccion.ID)
    if err != nil {
        panic("Error al actualizar el estado de la atracción: " + err.Error())
    }
}

// Si la atraccion se encuentra en el cuadrante abajo-izquierda
} else if atraccion.Posicionx >= 10 && atraccion.Posicionx <= 19 && atraccion.Posiciony >= 0 && atraccion.Posiciony <= 9 {

    // Comprobamos el clima de la ciudad asociada
    if ciudades[2].Temperatura < 20.0 || ciudades[2].Temperatura > 30.0 {

        // Actualizamos el estado de la atracción en la BD
        sentenciaPreparada, err := db.Prepare("UPDATE atraccion SET estado = ? WHERE id = ?")
        if err != nil {
            panic("Error al preparar la sentencia de modificación del estado de la atracción: " + err.Error())
        }

        defer sentenciaPreparada.Close()

        // Ejecutar sentencia, un valor por cada '?'
        _, err = sentenciaPreparada.Exec("Cerrada", atraccion.ID)
        if err != nil {
            panic("Error al actualizar el estado de la atracción: " + err.Error())
        }
    }

} else {
    // Actualizamos el estado de la atracción en la BD
    sentenciaPreparada, err := db.Prepare("UPDATE atraccion SET estado = ? WHERE id = ?")
    if err != nil {
        panic("Error al preparar la sentencia de modificación del estado de la atracción: " + err.Error())
    }

    defer sentenciaPreparada.Close()

    // Ejecutar sentencia, un valor por cada '?'
    _, err = sentenciaPreparada.Exec("Abierta", atraccion.ID)
    if err != nil {
        panic("Error al actualizar el estado de la atracción: " + err.Error())
    }
}

```

```

    // Si la atracción se encuentra en el cuadrante abajo-derecha
} else if atraccion.Posicionx >= 10 && atraccion.Posicionx <= 19 && atraccion.Posiciony >= 10 && atraccion.Posiciony <= 19 {

    // Comprobamos el clima de la ciudad asociada
    if ciudades[3].Temperatura < 20.0 || ciudades[3].Temperatura > 30.0 {

        // Actualizamos el estado de la atracción en la BD
        sentenciaPreparada, err := db.Prepare("UPDATE atraccion SET estado = ? WHERE id = ?")
        if err != nil {
            panic("Error al preparar la sentencia de modificación del estado de la atracción: " + err.Error())
        }

        defer sentenciaPreparada.Close()

        // Ejecutar sentencia, un valor por cada '?'
        _, err = sentenciaPreparada.Exec("Cerrada", atraccion.ID)
        if err != nil {
            panic("Error al actualizar el estado de la atracción: " + err.Error())
        }
    }

} else {
    // Actualizamos el estado de la atracción en la BD
    sentenciaPreparada, err := db.Prepare("UPDATE atraccion SET estado = ? WHERE id = ?")
    if err != nil {
        panic("Error al preparar la sentencia de modificación del estado de la atracción: " + err.Error())
    }

    defer sentenciaPreparada.Close()

    // Ejecutar sentencia, un valor por cada '?'
    _, err = sentenciaPreparada.Exec("Abierta", atraccion.ID)
    if err != nil {
        panic("Error al actualizar el estado de la atracción: " + err.Error())
    }
}

}

```

Abrimos la conexión con la BD, y recorremos las atracciones recibidas por parámetro para entonces dependiendo de su posición en el mapa, determinar su cuadrante y tras esto dependiendo del clima de la ciudad asociada a dicho cuadrante actualizamos el estado de la atracción a cerrada si la temperatura de la ciudad es menor a 20 grados o mayor a 30 grados, o a abierta en caso contrario.

Y con esto ya queda explicado todo el código del módulo engine.

Api_Engine

La función principal de este módulo es permitir la realización de peticiones Api Rest a nuestra aplicación de parque de atracciones, se puede realizar desde un front-end o desde otro software como puede ser el de Postman.

Cuenta con 4 manejadores para gestionar las peticiones API recibidas y posteriormente estas puedan devolver la información del parque en tiempo real.

- Visitantes: Obtiene información de los visitantes que se encuentran en el parque.
- Mapa: Obtiene la representación del mapa actualizado.
- Ciudades: Obtiene la información de las ciudades. Nombre y temperatura.
- Atracciones: Obtiene información de las atracciones del parque.

```
func main() {  
  
    ip := os.Args[1]  
    puerto := os.Args[2]  
  
    // IMPLEMENTAMOS EL API REST  
    // Rutas  
    mux := mux.NewRouter()  
  
    // Responder al cliente  
    mux.HandleFunc("/visitantes", getVisitantes).Methods("GET")  
    mux.HandleFunc("/mapa", getMapa).Methods("GET")  
    mux.HandleFunc("/ciudades", getCiudades).Methods("GET")  
    mux.HandleFunc("/atracciones", getAtracciones).Methods("GET")  
    // SERVIDOR  
    // Arrancamos el servidor https en una go routine  
    //go http.ListenAndServeTLS(":8081", "cert.pem", "key.pem", mux)  
    fmt.Println("Servidor API ENGINE corriendo en https://" + ip + ":" + puerto)  
    //log.Fatal(http.ListenAndServe(":8080", http.HandlerFunc(redirectToHttps)))  
    log.Fatal(http.ListenAndServe(": "+puerto, mux))  
}
```

La estructura de las peticiones para los 4 posibles casos es la siguiente.

```
http://localhost:8082/visitantes
```

Este es el formato de las funciones manejadoras, y todas son bastante similares por lo que vamos a comentar cómo se procede. La forma de obtener la información de los visitantes, ciudades, atracciones, etc. Es realizando una conexión con la base de datos de la aplicación y recuperando la información que corresponde, almacenando toda la información en una estructura que será convertida en Json.

```
/* Función manejadora para la obtención del estado de los visitantes */
func getVisitantes(rw http.ResponseWriter, r *http.Request) {
    log.Println("Petición de consulta del estado de los visitantes -> " + r.URL.Path)
    visitantes := []visitante{}

    // Accedemos a la base de datos, empezando por abrir la conexión
    db, err := sql.Open("mysql", "root:1234@tcp(127.0.0.1:3306)/parque_atracciones")

    // Comprobamos que no haya error al conectarse
    if err != nil {
        panic("Error al conectarse con la BD: " + err.Error())
    }

    defer db.Close() // Para que siempre se cierre la conexión con la BD al finalizar el programa

    rows, err := db.Query("SELECT id, nombre, posicionx, posiciony, destinox, destinoy, idEnParque, ultimoEvento FROM visitante")
    // Comprobamos que no se produzcan errores al hacer la consulta
    if err != nil {
        panic("Error al consultar el estado de los visitantes en la BD: " + err.Error())
    }

    defer rows.Close()

    for rows.Next() {
        v := visitante{}
        rows.Scan(&v.ID, &v.Nombre, &v.Posicionx, &v.Posiciony, &v.Destinox, &v.Destinoy, &v.IdEnParque, &v.UltimoEvento)
        visitantes = append(visitantes, v)
    }

    // CONTINUAR IMPLEMENTACIÓN
    SendDataGetVisitantes(rw, visitantes)
}
```

En este caso por ejemplo, almacenamos la información de los visitantes, de los cuales recuperamos la siguiente información.

- ID
- Nombre
- Posicionx
- Posiciony
- Destinox
- Destinoy
- IdEnParque
- UltimoEvento

La estructura que devuelven las distintas peticiones tienen el siguiente formato.

- Data : Almacena la información en formato Json.
- Status : Información si la petición ha sido valida.
- Message : Mensaje asociado al status
- ContentType : Tipo de contenido

```
// Estructura con el formato de una respuesta http
type Response struct {
    Status      int      `json:"status"`
    Data        interface{} `json:"data"`
    Message     string   `json:"message"`
    contentType string
    responseWrite http.ResponseWriter
}

/* Función que crea una respuesta por defecto para los clientes de la API REST */
func CreateDefaultResponse(rw http.ResponseWriter) Response {
    return Response{
        Status:          http.StatusOK,
        responseWrite: rw,
        contentType:    "application/json",
    }
}

/* Función que envía las respuestas a los clientes de la API REST */
func (resp *Response) Send() {
    resp.responseWrite.Header().Set("Content-Type", resp.contentType)
    resp.responseWrite.Header().Set("Access-Control-Allow-Origin", "*")
    resp.responseWrite.WriteHeader(resp.Status)

    // Marshall devuelve 2 valores: Los valores transformados en tipo byte y un error
    output, _ := json.Marshal(&resp) // Para responder con json
    //output, _ := xml.Marshal(&resp) // Para responder con xml
    //output, _ := yaml.Marshal(&resp) // Para responder con yaml
    fmt.Fprintln(resp.responseWrite, string(output))
}
```

En la siguiente imagen podemos ver cómo están implementadas las peticiones.

Se almacena la información en una variable response, el cual cuenta con el campo Data donde se almacena la información de la correspondiente petición, un message donde se confirma que la petición ha sido ejecutada correctamente y finalmente se procede a enviarse con la función Send().

```
/* Función que envía una respuesta a los clientes indicando que la consulta de las atracciones ha sido satisfactoria */
func SendDataGetAtracciones(rw http.ResponseWriter, data interface{}) {
    response := CreateDefaultResponse(rw)
    response.Data = data
    response.Message = "OK: Estado actual de las atracciones obtenido."
    response.Send()
}

/* Función que envía una respuesta a los clientes indicando que el registro ha sido satisfactorio */
func SendDataGetMapa(rw http.ResponseWriter, data interface{}) {
    response := CreateDefaultResponse(rw)
    response.Data = data
    response.Message = "OK: Estado actual del mapa obtenido."
    response.Send()
}

/* Función que envía una respuesta a los clientes indicando que el registro ha sido satisfactorio */
func SendDataGetCiudades(rw http.ResponseWriter, data interface{}) {
    response := CreateDefaultResponse(rw)
    response.Data = data
    response.Message = "OK: Información de las ciudades obtenida."
    response.Send()
}

func (resp *Response) NotFound() {
    resp.Status = http.StatusNotFound
    resp.Message = "ERROR: Resource Not Found"
}
```

Front-End

Está hecho en con React, esta librería nos permite crear componentes para representar la información del parque.

En la parte de App importamos los distintos componentes necesarios para que se visualice la web.

```
import React from 'react';
import './App.css';

import "primereact/resources/themes/lara-light-indigo/theme.css";
import "primereact/resources/primereact.min.css";
import "primeicons/primeicons.css";
    //icons

//import Menu from './componentes/menu/Menu';
//import Footer from './componentes/footer/Footer'
import Mapa from './componentes/mapa/Mapa'
import Visitante from './componentes/visitantes/Visitantes'
import Ciudad from './componentes/ciudades/Ciudades'
import Atraccion from './componentes/atracciones/Atracciones'

function App() {
  return (
    <div className="App">
      <Mapa />
      <Atraccion />
      <Visitante />
      <Ciudad />
    </div>
  );
}

export default App;
```

En la siguiente captura declaramos el constructor de la clase donde tenemos las variables visitantes que contendrá la información de la API y la propiedad isFetch cuyo valor es booleano y sirve en caso de que la petición no se haya realizado con éxito nos muestre una pantalla de carga.

En el componentDidMount el código se lanza después del renderizado del componente y nos sirve para realizar la petición Api con fetch, la petición será devuelta en formato Json y la almacenamos en la variable visitantes y isFetch pasa a ser false.

```
class Visitantes extends React.Component {
    /**
     * Constructor de la clase Visitantes
     * @param {} props
     */
    constructor(props) {
        super(props)
        this.state = {
            visitantes: [],
            isFetch: true,
        }
    }
    /**
     * ComponentDidMount que carga la información de los visitantes
     */
    componentDidMount() {
        fetch("http://localhost:8082/visitantes")
            .then(response => response.json())
            .then(visitantesJson => this.setState( {
                visitantes: visitantesJson.data,
                isFetch: false
            }))
            .catch(error => console.log(error))
    }
}
```

En el render mostramos por pantalla la información de los visitantes, ciudades y atracciones. Esta información está contenida en un DataTable donde obtenemos las columnas de la entidad almacenadas en la base de datos.

```
/**  
 * Render que muestra la información de los visitantes  
 * @returns : Renderizado de los visitantes  
 */  
render () {  
  
    const { visitantes, isFetch } = this.state  
  
    if (isFetch) {  
        return <div>El estado de los visitantes no está disponible por el momento</div>  
    }  
    return (  
        <div className ="container">  
            <DataTable value={visitantes}>  
                <Column field="id" header="ID"></Column>  
                <Column field="nombre" header="Nombre"></Column>  
                <Column field="posicionx" header="Posición x"></Column>  
                <Column field="posiciony" header="Posición y"></Column>  
                <Column field="destinox" header="Destino x"></Column>  
                <Column field="destinoy" header="Destino y"></Column>  
                <Column field="idEnParque" header="ID en el parque"></Column>  
                <Column field="ultimoEvento" header="Log"></Column>  
            </DataTable>  
        </div>  
    );  
}
```

Guía de despliegue de la aplicación

Lo primero que haremos será conectarnos a una red fiable, en nuestro caso el despliegue lo haremos utilizando uno de nuestros móviles como router ya que en la universidad la conectividad está capada.

Después comprobaremos las ips de cada una de las máquinas que van a ser utilizadas para el experimento:

IP PC WILMER -> 192.168.43.201
IP PC JOSE -> 192.168.43.50
IP MÁQUINA VIRTUAL -> 192.168.43.247

Los puertos utilizados son los siguientes:

- 9092 -> Gestor de colas
- 9093 -> Registry vía sockets
- 8081 -> Registry vía api rest
- 9094 -> Servidor de tiempos de espera
- 8082 -> Servidor API_Engine
- 3000 -> Front-End

En golang para poner en marcha alguna aplicación tenemos que ir a la carpeta de cada uno de los módulos.

Para compilar ejecutamos un comando como por ejemplo este:

```
go build fwq_engine.go
```

Finalmente, para ejecutar un programa escribimos un comando de este estilo:

```
./fwq_engine 127.0.0.1 9092 30 127.0.0.1 9094
```

Entonces tenemos lo siguiente, donde cada módulo lo ejecutamos desde su respectivo directorio asociado, es decir, fwq_engine se encuentra en el directorio FWQ_ENGINE.

PC WILMER (Gestor Colas, FWQ_WaitingTimeServer, FRONT) - 192.168.43.201

Modificamos los ficheros de configuración de kafka y le asignamos la ip 192.168.43.201 y el puerto 9092.

```
bin/zookeeper-server-start.sh config/zookeeper.properties (desde el directorio de kafka)
```

```
bin/kafka-server-start.sh config/server.properties (desde el directorio de kafka)
```

```
./fwq_waitingTimeServer 192.168.43.201 9094 192.168.43.201 9092
```

/FRONT/npm start -> Lanza la aplicación web en 192.168.43.201:3000

PC JOSE (API_Engine, FWQ_Engine y FWQ_Registry) - 192.168.43.50

```
./api_engine 192.168.43.50 8082
```

```
./fwq_engine 192.168.43.201 9092 2 192.168.43.201 9094
```

```
./fwq_registry 192.168.43.50 9093
```

MÁQUINA VIRTUAL (FWQ_Visitor (1 a n), FWQ_Sensor (1 a n)) - 192.168.43.247

```
./fwq_visitor 192.168.43.50 9093 8081 192.168.43.201 9092
```

```
./fwq_sensor 192.168.43.201 9092 atraccion(1-16)
```

Capturas de pantalla que muestran el funcionamiento de las distintas aplicaciones conectadas

En este apartado final de la memoria vamos a mostrar capturas del funcionamiento de los distintos módulos en funcionamiento para demostrar el correcto proceder de estos.

Captura de ejecución de un sensor en la atracción 15:

```
jose@jose-B450-AORUS-PRO:~/go/src/github.com/wilmer799/SDPracticas/FWQ_Sensor$ ./script.sh
Sensor creado para la atracción (atraccion15) en la que inicialmente hay 39 personas en cola
2022/06/19 21:40:00 En la atracción [atraccion15] hay 25 personas en cola
2022/06/19 21:40:04 En la atracción [atraccion15] hay 38 personas en cola
2022/06/19 21:40:08 En la atracción [atraccion15] hay 54 personas en cola
2022/06/19 21:40:12 En la atracción [atraccion15] hay 16 personas en cola
2022/06/19 21:40:16 En la atracción [atraccion15] hay 53 personas en cola
2022/06/19 21:40:20 En la atracción [atraccion15] hay 38 personas en cola
2022/06/19 21:40:24 En la atracción [atraccion15] hay 40 personas en cola
2022/06/19 21:40:28 En la atracción [atraccion15] hay 16 personas en cola
2022/06/19 21:40:32 En la atracción [atraccion15] hay 27 personas en cola
2022/06/19 21:40:36 En la atracción [atraccion15] hay 45 personas en cola
2022/06/19 21:40:40 En la atracción [atraccion15] hay 49 personas en cola
2022/06/19 21:40:44 En la atracción [atraccion15] hay 21 personas en cola
2022/06/19 21:40:48 En la atracción [atraccion15] hay 0 personas en cola
2022/06/19 21:40:52 En la atracción [atraccion15] hay 37 personas en cola
2022/06/19 21:40:56 En la atracción [atraccion15] hay 8 personas en cola
2022/06/19 21:41:00 En la atracción [atraccion15] hay 54 personas en cola
2022/06/19 21:41:04 En la atracción [atraccion15] hay 59 personas en cola
2022/06/19 21:41:08 En la atracción [atraccion15] hay 44 personas en cola
2022/06/19 21:41:12 En la atracción [atraccion15] hay 57 personas en cola
2022/06/19 21:41:16 En la atracción [atraccion15] hay 56 personas en cola
2022/06/19 21:41:20 En la atracción [atraccion15] hay 10 personas en cola
```

Captura de funcionamiento del servidor de tiempos:

```
jose@jose-B450-AORUS-PRO:~/go/src/github.com/wilmer799/SDPracticas/FWQ_WaitingTimeServer$ ./script.sh
Servidor de tiempos atendiendo en localhost:9094
[ atraccion15:38 personas en cola ]
[ atraccion15:40 personas en cola ]
[ atraccion15:16 personas en cola ]
[ atraccion15:27 personas en cola ]
[ atraccion15:45 personas en cola ]
[ atraccion15:49 personas en cola ]
[ atraccion15:21 personas en cola ]
[ atraccion15:0 personas en cola ]
[ atraccion15:37 personas en cola ]
[ atraccion15:8 personas en cola ]

2022/06/19 21:40:57 Cliente engine 127.0.0.1:37372 conectado.

2022/06/19 21:40:57 Petición de un engine recibida.

Estado actual de las atracciones:
ID:TCiclo:NVisitantes:TiempoEspera
atraccion1:5:3:45
atraccion10:15:4:13
atraccion11:17:7:7
atraccion12:7:8:17
atraccion13:8:2:77
atraccion14:12:4:40
atraccion15:13:10:74
atraccion16:19:11:23
atraccion2:8:9:30
atraccion3:7:6:15
atraccion4:18:9:65
atraccion5:4:5:10
atraccion6:10:6:40
atraccion7:11:8:80
atraccion8:19:7:90
atraccion9:14:6:20

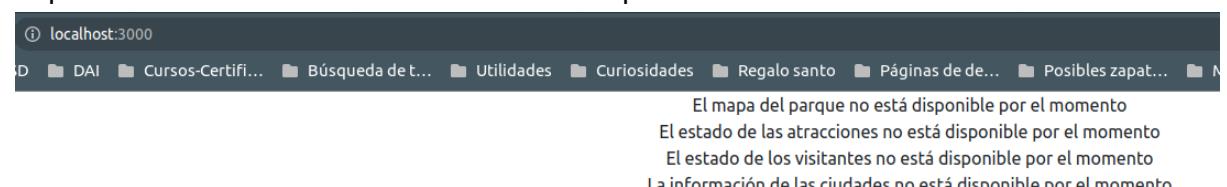
Enviando los tiempos de espera actualizados...

2022/06/19 21:40:57 Engine127.0.0.1:37372 desconectado.
```

Captura de funcionamiento del registry ante dos peticiones de registro, una por sockets y la otra por api rest:

```
jose@jose-B450-AORUS-PRO:~/go/src/github.com/wilmer799/SDPracticas/FWQ_Registry$ ./script.sh
Arrancando el Registry, atendiendo vía sockets en localhost:9093
Servidor API REST corriendo en https://localhost:8081
2022/06/19 21:45:20 Visitante 127.0.0.1:33960 conectado.
Visitante a registrar -> ID: jose123 | Nombre: jose | Password: 1234
Registro completado. Actualmente hay 1 visitantes registrados.
2022/06/19 21:45:25 Visitante 127.0.0.1:33960 desconectado.
2022/06/19 21:45:34 Petición de creación de perfil recibida -> /crear/carlos123
Visitante a registrar -> ID: carlos123 Nombre: carlos Password: 1234
Registro completado. Actualmente hay 2 visitantes registrados.
```

Captura del estado del Front antes de estar disponible la información:



Captura del estado actual del engine antes de recibir peticiones de entrada al parque:

```
***** FUN WITH QUEUES RESORT ACTIVITY MAP *****
ID           Nombre      Pos.        Destino      DentroParque
carlos123    carlos     (0,0)       (-1,-1)     0
jose123      jose      (0,0)       (-1,-1)     0

***Conexión con el servidor de tiempo de espera***

***Actualizando los tiempos de espera***

Enviando información de las atracciones...
2022/06/19 21:49:02 Tiempos de espera actualizados: atraccion1:45|atraccion10:13|atraccion4:65|atraccion5:10|atraccion6:40|atraccion7:80|atraccion8:90|atraccion9:20| 

La ciudad del cuadrante arriba-izquierda es: Berlin y su temperatura es: 19.45 °C
La ciudad del cuadrante arriba-derecha es: New York y su temperatura es: 22.79 °C
La ciudad del cuadrante abajo-izquierda es: London y su temperatura es: 14.8 °C
La ciudad del cuadrante abajo-derecha es: Coruña y su temperatura es: 14.13 °C
```

Al recargar el front se realizan las peticiones al api_engine:

```
jose@jose-B450-AORUS-PRO:~/go/src/github.com/wilmer799/SDPracticas/API_Engine$ ./script.sh

Servidor API ENGINE corriendo en https://localhost:8082
2022/06/19 21:49:52 Petición de consulta del estado del mapa -> /mapa
2022/06/19 21:49:52 Petición de consulta del estado de las atracciones -> /atracciones
2022/06/19 21:49:52 Petición de consulta del estado de los visitantes -> /visitantes
2022/06/19 21:49:52 Petición de consulta de las ciudades -> /ciudades
2022/06/19 21:49:52 Petición de consulta del estado de las atracciones -> /atracciones
2022/06/19 21:49:52 Petición de consulta del estado del mapa -> /mapa
2022/06/19 21:49:52 Petición de consulta del estado de los visitantes -> /visitantes
2022/06/19 21:49:52 Petición de consulta de las ciudades -> /ciudades
```

Y ahora ya podemos consultar la información actual del parque:

Fila	InfoParque
0	--- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
1	- --- --- --- --- --- --- --- --- --- --- --- --- ---
2	- --- --- (30) --- --- --- --- --- --- --- ---
3	- --- --- (90) --- --- --- --- --- --- ---
4	- --- --- --- --- --- --- --- --- --- --- --- --- --- 40 ---
5	- --- --- --- --- --- --- --- --- --- --- --- --- 0 ---
6	- --- --- --- --- --- --- --- --- --- --- --- --- ---
7	- --- --- --- (17) (15) --- --- --- --- ---
8	- --- --- --- --- (20) --- --- --- ---
9	- --- --- --- --- --- --- --- --- --- --- ---
10	- --- (80) --- --- --- --- --- --- --- 10 ---
11	- --- --- --- --- --- --- --- --- (45) --- --- (65)
12	- --- --- --- --- --- --- --- --- --- --- ---
13	- --- --- --- --- --- --- --- --- --- ---
14	- --- --- --- --- --- --- --- --- --- ---
15	- --- --- --- --- --- --- --- --- --- ---
16	- --- --- --- --- --- --- --- --- (23) ---
17	- --- --- --- --- --- --- --- --- --- (77) ---
18	- --- --- --- (7) --- --- --- --- ---
19	- --- --- --- --- (13) --- --- ---

Démonos cuenta como en el mapa aún no hay ningún visitante.

ID	Tiempo de ciclo	Capacidad de visitantes	Posición x	Posición y	Tiempo de espera en seg	Estado
atraccion1	5	3	10	14	45	Cerrada
atraccion10	15	4	18	11	13	Cerrada
atraccion11	17	7	17	5	7	Cerrada
atraccion12	7	8	6	5	17	Cerrada
atraccion13	8	2	16	17	77	Cerrada
atraccion14	12	4	19	18	40	Cerrada
atraccion15	13	10	4	15	0	Abierta
atraccion16	19	11	15	15	23	Cerrada
atraccion2	8	9	1	4	30	Cerrada
atraccion3	7	6	6	6	15	Cerrada
atraccion4	18	9	10	19	65	Cerrada
atraccion5	4	5	9	17	10	Abierta
atraccion6	10	6	3	18	40	Abierta
atraccion7	11	8	9	2	80	Cerrada
atraccion8	19	7	2	3	90	Cerrada
atraccion9	14	6	7	8	20	Cerrada

ID	Nombre	Posición x	Posición y	Destino x	Destino y	ID en el parque	Log
carlos123	carlos	0	0	-1	-1	c	2022-06-19 21:45:34 127.0.0.1:45764 Alta Visitante carlos123 registrado correctamente
jose123	jose	0	0	-1	-1	j	2022-06-19 21:45:25 127.0.0.1:33960 Alta Visitante jose123 registrado correctamente
Cuadrante			Nombre			Temperatura	
abajo-derecha			Coruña			14.09	
abajo-izquierda			London			14.8	
arriba-derecha			New York			22.85	
arriba-izquierda			Berlin			19.09	

Podemos ver ya en el front toda la información disponible sobre atracciones, visitantes, ciudades y mapa. Vamos pues a meter a los visitantes en el parque.

Nota: Si recargamos la página podemos ver cómo se va actualizando el tiempo de espera de la atracción 15:

atraccion15	13	10	4	15	13	Abierta
-------------	----	----	---	----	----	---------

Captura de funcionamiento del visitante con las dos peticiones de registro mencionadas:

```
jose@jose-B450-AORUS-PRO:~/go/src/github.com/wilmer799/SDPracticas/FWQ_Visitor$ ./script.sh
Creado un visitante que envía peticiones a un registry por localhost:9093/8081 y a un engine por lo
calhost:9092

**Bienvenido al parque de atracciones**

***Menu parque de atracciones***
1.Crear perfil
2.Editar perfil
3.Moverse por el parque
Elige la acción a realizar:1

*****Creación de perfil*****

Selecciona el tipo de conexión al registry:
1 -> Sockets
2 -> API_REST

1

Introduce tu ID:jose123
Introduce tu nombre:jose
Introduce tu contraseña:1234
2022/06/19 21:45:25 Respuesta del Registry: Visitante registrado en el parque.

***Menu parque de atracciones***
1.Crear perfil
2.Editar perfil
3.Moverse por el parque
Elige la acción a realizar:1

*****Creación de perfil*****


Selecciona el tipo de conexión al registry:
1 -> Sockets
2 -> API_REST

2

Introduce tu ID:carlos123
Introduce tu nombre:carlos
Introduce tu contraseña:1234

{"status":200,"data":null,"message":"OK: Visitante registrado correctamente"}

***Menu parque de atracciones***
1.Crear perfil
2.Editar perfil
3.Moverse por el parque
Elige la acción a realizar:[]
```

Captura del visitante y el registry con una petición de modificación y otra de entrada al parque:

Captura del engine respectiva:

```
***** FUN WITH QUEUES RESORT ACTIVITY MAP *****
ID           Nombre      Pos.        Destino      DentroParque
carlos123    carlos      (0,0)       (-1,-1)     0
jose123      joselito   (8,17)      (9,17)      1

***Conexión con el servidor de tiempo de espera***

***Actualizando los tiempos de espera***

Enviando información de las atracciones...
2022/06/19 21:57:01 Tiempos de espera actualizados: atraccion1:45|atraccion10:13|atraccion4:65|atraccion5:10|atraccion6:40|atraccion7:80|atraccion8:90|atraccion9:20| 

La ciudad del cuadrante arriba-izquierda es: Berlin y su temperatura es: 19.09 °C
La ciudad del cuadrante arriba-derecha es: New York y su temperatura es: 22.83 °C
La ciudad del cuadrante abajo-izquierda es: London y su temperatura es: 14.78 °C
La ciudad del cuadrante abajo-derecha es: Coruña y su temperatura es: 13.88 °C

Local instance 3306
***** FUN WITH QUEUES RESORT ACTIVITY MAP *****
ID           Nombre      Pos.        Destino      DentroParque
carlos123    carlos      (0,0)       (-1,-1)     0
jose123      joselito   (8,17)      (9,17)      1

Petición recibida: jose123:S:-1,-1
Petición recibida: jose123:NE:3,18

***Conexión con el servidor de tiempo de espera***

***Actualizando los tiempos de espera***

Enviando información de las atracciones...
2022/06/19 21:57:06 Tiempos de espera actualizados: atraccion1:45|atraccion10:13|atraccion4:65|atraccion5:10|atraccion6:40|atraccion7:80|atraccion8:90|atraccion9:20| 

La ciudad del cuadrante arriba-izquierda es: Berlin y su temperatura es: 19.09 °C
La ciudad del cuadrante arriba-derecha es: New York y su temperatura es: 22.83 °C
La ciudad del cuadrante abajo-izquierda es: London y su temperatura es: 14.78 °C
La ciudad del cuadrante abajo-derecha es: Coruña y su temperatura es: 13.88 °C

***** FUN WITH QUEUES RESORT ACTIVITY MAP *****
ID           Nombre      Pos.        Destino      DentroParque
carlos123    carlos      (0,0)       (-1,-1)     0
jose123      joselito   (8,18)      (3,18)      1

Petición recibida: jose123:N:3,18
Petición recibida: jose123:N:3,18
```

Vemos cómo se van recibiendo los movimientos y cómo se va actualizando el estado del visitante.

Por último si consultamos el front podemos ver la información actualizada ante esta situación:

Fila	InfoParque
0	- --- --- --- --- --- --- --- --- ---
1	- ----- ----- ----- ----- -----
2	- - - (30) ----- ----- ----- -----
3	- - (90) ----- ----- ----- -----
4	- ----- ----- ----- ----- 40 -
5	- ----- ----- ----- ----- 26 -----
6	- ----- ----- ----- ----- -----
7	- - - (17) (15) ----- ----- -----
8	- ----- (20) ----- ----- -----
9	- ----- ----- ----- ----- -----
10	- (80) ----- ----- ----- 10 -----
11	- ----- ----- ----- ----- (45) ----- (65)
12	- ----- ----- ----- ----- -----
13	- ----- ----- ----- ----- -----
14	- ----- ----- ----- ----- -----
15	- ----- ----- ----- ----- -----
16	- ----- ----- ----- ----- (23) -----
17	- ----- ----- ----- ----- (77) -----
18	- - - (7) ----- ----- ----- -----
19	- ----- ----- ----- (13) ----- -----

El visitante se encuentra en el parque, concretamente en la fila 8.

ID	Nombre	Posición x	Posición y	Destino x	Destino y	ID en el parque	Log
carlos123	carlos	0	0	-1	-1	c	2022-06-19 21:45:34 127.0.0.1:45764 Alta Visitante carlos123 registrado correctamente
jose123	joselito	6	18	3	18	j	2022-06-19 21:56:08 127.0.0.1:45766 Modificación Visitante jose123 actualizado correctamente

Y aquí podemos ver también la información de dicho visitante actualizada.