Sistemas Distribuidos

2020/2021

Práctica II

"Fun with Queues"

Universitat d'Alacant
Universidad de Alicante

Elvi Mihai Sabau Sabau - 51254875L

Iván Sabater Domínguez - 48797056A

Fecha: 30/10/2021

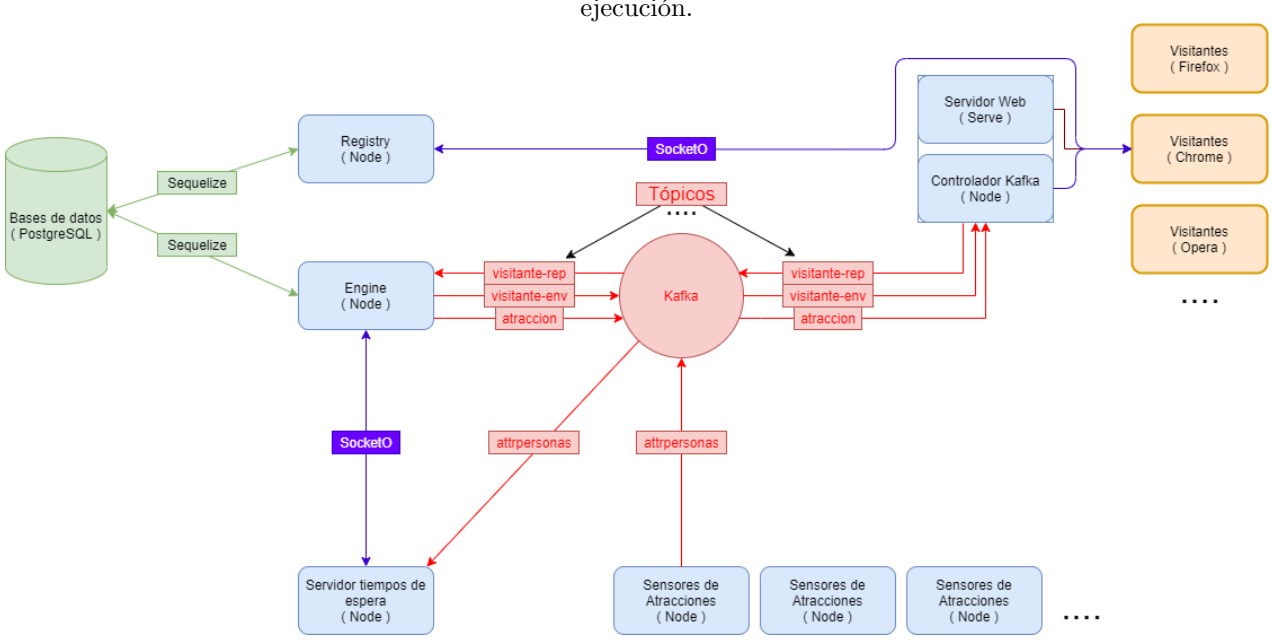
Índice

1. Tecnologías usadas

- **PostgreSQL**: Se ha usado PostgreSQL para la base de datos.
- **Kafka**: Se ha usado Kafka como servidor de mensajería e intermediario entre aplicaciones.
- **JavaScript & Node**: Se ha usado el lenguaje de programación JavaScript, y el entorno de ejecución Node.
- **Sequelize**: Se ha usado Sequelize como librería ORM para el control de transacciones entre la aplicación y la base de datos.
- **Bcrypt**: Se ha usado la librería Bcrypt para "Saltear" las contraseñas de los usuarios en la base de datos, de esta manera nunca se expone la contraseña en texto plano en el sistema.
- **Socket.IO**: Se ha usado la librería Socket.IO para el control y flujo de datos de sockets entre cliente - servidor.
- **Socket.IO-encrypt**: Se ha usado esta librería middleware en conjunto con Socket.IO para encriptar la transmisión de datos entre cliente - servidor usando un "Secret", usando un encriptado simétrico (el mismo secret sirve tanto para encriptar como para desencriptar, en ningún momento dicho secret se transmite por un medio, y ambas partes conocen dicho secret de antemano).
- **kafka-node**: Se ha usado la librería kafka-node para el control de mensajes y conexión con el servidor de Kafka.
- **ReactJS**: Se ha usado React como framework front-end para el desarrollo de las interfaces de usuario usando páginas web.
- **Concurrently**: Se ha usado la librería Concurrently para la ejecución de dos comandos simultáneos independientemente del sistema operativo en el que se ejecuta, este se usa para ejecutar en el mismo contenedor Serve y el controlador de Kafka para los Visitantes.
- **Serve**: Se ha usado la librería Serve para crear un pequeño servidor web el cual sirve la página web creada con React.
- **Docker & docker-compose**: Se ha usado el sistema de encapsulación y contenerización Docker para el despliegue del sistema de manera distribuida, y docker-compose para desplegar Kafka y la Base de datos Postgres.
- **Git & GitHub**: Se ha usado Git y GitHub para un sistema de control de versiones y desarollo remoto en equipo.

2. Estructura del Sistema

La estructura de nuestro sistema sería el siguiente, usando las tecnologías mencionadas en el diagrama para su ejecución.



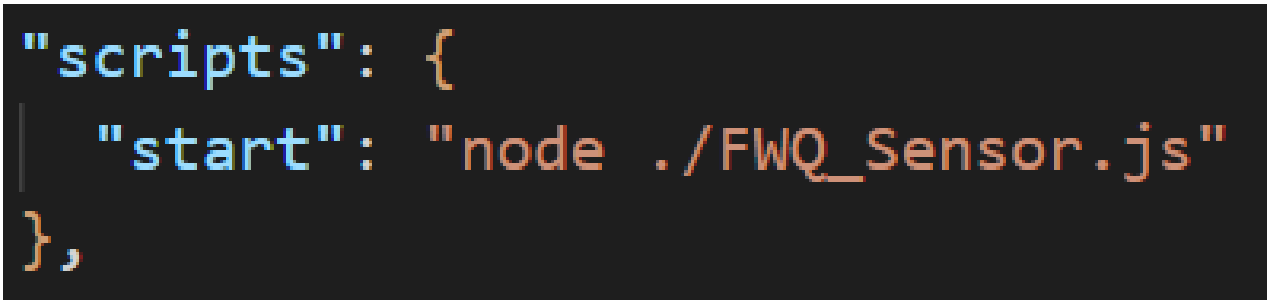
2.1. Dependencias

Las dependencias generales del sistema son:

- Windows, Ubuntu o derivados de GNU/Linux.
- Node v16.13.0.
- Docker 20.10.8.
- docker-compose 1.25.0.

2.2. Script

Hemos creado un script de lanzamiento en nuestro package.json para cada módulo de la aplicación para facilitar y estandarizar el lanzamiento de la aplicación durante el desarollo.



De esta manera en vez que tener que, por ejemplo en el sensor, ejecutar el comando:

```
node ./FWQ_Sensor.js [parámetros]
```

Vamos a usar en todos los módulos:

```
npm start [parámetros]
```

Este script se usa tanto en producción como en despliegue dentro de los contenedores, nosotros a la hora de desplegar no hará falta que lo arranquemos de esta manera, ya que usamos docker, pero lo explicamos igualmente para mencionar como arranca internamente cada modulo dentro del contenedor.

2.3. Registry

Nuestro módulo Registry se encarga de la gestión de las cuentas de los usuarios, se interconecta vía Sockets con los navegadores de los visitantes, y recibe los datos de los usuarios.

Permite tanto el registro como la modificación de las cuentas como la autenticación y desautenticación de estos.

Hemos decidido realizar la lógica de sesiones en este módulo por 2 motivos mayores:

1. Para evitar que el sistema tenga lógica desorganizada, ya que debido a que el registry se encarga obligatoriamente del registro, hemos decidido que además se encargará de la gestión de las sesiones, de esta manera tendremos el sistema menos cohesionado y evitaremos ciertos patrones de mala práctica que se han enseñado en teoría.
2. El uso de sockets nos proporcionará una gestión más limpia y control de las sesiones, ya que detectaremos al momento cuando alguien cierra la página web, de esta manera podremos desautenticar dicho usuario sin necesidad de usar más lógica de control (por ejemplo, revisar cada 4 segundos si el usuario "sigue vivo").

El módulo, además, a la hora de registrar y modificar las cuentas, se encriptan las contraseñas con un sistema de salteo usando un encriptado unidireccional, guardando la contraseña encriptada en la base de datos.

A la hora de iniciar sesión, el Registry recibe la contraseña del usuario, la encripta, y la compara con la versión previamente encriptada guardada en la base de datos.

No hay que olvidar que no solo esta, sino, todas las conexiones vía socket están cifradas con un encriptado simétrico usando un Secret que se pasa como parámetro al desplegar.

De esta manera añadimos unas capas extra de seguridad sobre nuestra aplicación.

Si el registry cae, el visitor intetará reconectarse cada cierto tiempo.

2.4. Visitor

El módulo visitor está formado por 3 partes, debido a que nuestro cliente no es una app de escritorio, sino, una página web, a la cual, todos pueden acceder.

2.4.1. Servidor web

El servidor web usado "Serve" es un sencillo servidor que sirve archivos de manera estática, nosotros lo usamos para servir nuestra página web (interfaz) a los visitantes de nuestro parque de atracciones.

2.4.2. Controlador Kafka

Tenemos aparte un proceso en Node que se encarga de controlar las conexiones entre los navegadores y el servidor de Kafka usando sockets, debido a que no es posible ejecutar un cliente de Kafka en el navegador, pero si es posible usar un middleware, muchos usan una REST API o un Kafka WebSocket. Nosotros usamos lo segundo, siendo nuestro Controlador Kafka un "Kafka WebSocket".

Por cada conexión que recibe nuestro controlador kafka de los navegadores, este crea una nueva conexión con el servidor de kafka, generando el productor y los consumidores necesarios para la transmisión de datos.

- *Consumidor de Atracciones*: Este consumidor recibe desde el servidor Kafka y este desde el Engine, un diccionario con las atracciones actuales, sus coordenadas, tiempo de espera de la cola, y una imágen (optativa). Usa el tópico "atraccion".
- *Consumidor de Usuarios*: Este consumidor recibe desde el servidor Kafka y este desde el Engine cada vez que un usuario hace un movimiento en el mapa, se reciben sus datos, sus coordenadas actuales, y su destino. Usa el tópico "visitante-env".
- *Productor de Usuario*: Este productor envía los datos del usuario conectado desde el navegador al servidor de Kafka y este al Engine, cada vez que el usuario realiza un movimiento. Usa el tópico "visitante-rep".

Ya que el controlador y el servidor se ejecutarán en la misma máquina docker, se usa concurrently para la ejecución concurrente de ambos procesos.

2.4.3. Navegador web

El navegador recibe la página web desde el Servidor web, la renderiza, y realiza la lógica descrita en el código JavaScript.

Realiza la lógica de conexión con el Controlador Kafka y la lógica de movimiento y decisión a que atracción ir, dependiendo del tiempo de espera.

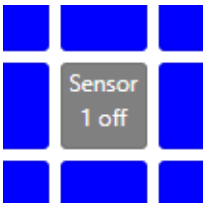
Además de mostrar los formularios, interactuar y enviar los datos de registro, autenticación y modificación de perfil al Registry.

Si el Registry cae, el navegador redirigirá al usuario a la página principal, e intentará reconectarse, cada cierto tiempo.

Si durante el camino, la atracción destino supera los 60 min de tiempo de espera en cola, el cliente buscará otra atracción a la cual ir. Y si todas las atracciones superan dicho tiempo, el cliente se quedará en el sitio esperando.

Cuando el usuario cierra sesión (sale del parque) sus coordenadas actuales cambian a las de la entrada del parque, lo mismo sucede si el usuario cierra la página web inesperadamente.

También interesa mencionar que si el waiting time server, o el engine, o alguno de los sensores deja de funcionar, en el mapa se mostrara dicho sensor en gris, mostrando el texto "sensor X off".



2.5. Sensor

Cada sensor define una atracción en el parque.

El sensor manda vía kafka usando un productor, una ID de atracción, unas coordenadas X e Y, un número aleatorio de personas, y la imágen de la atracción (optativa) al Waiting Time Server. Usa el tópico “attrpersonas”.

En caso de que un sensor se apague, en el mapa se mostrará en gris, y en caso de reiniciarse con unas coordenadas nuevas, el sensor se actualizará de posición en el mapa.

Tambien interesa decir, que si dicho sensor (identificado por la ID) vuelve a arrancar, pero con otras coordenadas, este se rehubica en el mapa.

2.6. Waiting Time Server

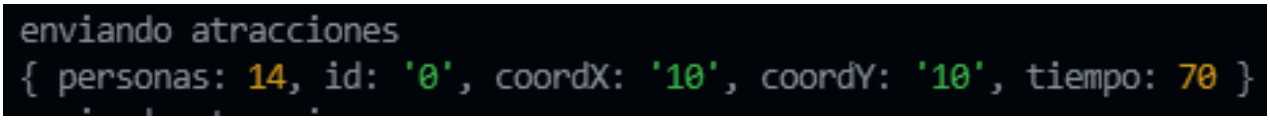
El servidor de tiempos de espera, recibe los datos vía cliente kafka usando un consumidor los datos enviados por los sensores. Usa el tópico “attrpersonas”.

Recibe los datos de la atracción, calcula el tiempo estimado dependiendo de las personas en la cola, y los guarda en un diccionario, usando como valor la ID de la atracción.

De esta manera, si el servidor recibe datos actualizados de la misma atracción, sólo se actualizará su posición en dicho diccionario de atracciones.

Así, este diccionario siempre tendrá una posición unica de cada una de las atracciones actualizadas.

El servidor sirve un servidor de sockets, esperando la conexión del engine, cuando este recibe dicha conexión, el Engine puede solicitar en cualquier momento el diccionario de atracciones.



2.7. Engine

El engine es la pieza central de nuestro sistema.

Recibe los datos de los usuarios vía cliente kafka con un consumidor sus posiciones actualizadas, las actualiza en la base de datos, y las vuelve a mandar de vuelta a los visitantes usando un productor. Usan el tópicos “visitante-env, visitante-rep”.

```
Dato recibido de Usuario {
  id: 1,
  name: 'a',
  password: '$2b$10$QCGCT8/nzK7sdWGSSjOUQu2MbYm2QTwSYz1p/GIN2F3zIbYaw0R2K',
  x_actual: 9,
  y_actual: 9,
  x_destino: 9,
  y_destino: 9,
  logged: true,
  createdAt: '2021-10-24T13:20:38.527Z',
  updatedAt: '2021-10-24T13:20:39.935Z'
}
{ 'visitante-env': { '0': 12385 } }
```

Además, por cada dato recibido de un usuario, envia a los visitantes el diccionario de atracciones, este lo obtiene vía conexión socket desde el Waiting Time Server, el diccionario de atracciones actualizadas. y las envía a los visitantes vía cliente kafka con un productor, Usa el tópico “atraccion”.

```
solcitando atracciones
atracciones recibidas [
  { personas: 0, id: '0', coordX: '3', coordY: '3', tiempo: 0 },
  { personas: 10, id: '1', coordX: '18', coordY: '2', tiempo: 50 }
]
```

Si el Waiting Time Server cae, el Engine intentará reconectarse cada cierto tiempo.

2.8. Desplegando la BD y Kafka

Antes de arrancar nuestros módulos en produccion o desarollo, vamos a desplegar la base de datos y el servidor de kafka, para ello hemos facilitado un archivo "docker-compose.yml".

Este archivo ejecutado con docker-compose arrancará 4 contenedores.

- Una base de datos PostgreSQL con el usuario 'root' y contraseña 'root'
- Un ligero cliente de DB servido en una página web llamado Adminer.
- Un servidor Zookeeper.
- Un servidor de Kafka.

Antes de ejecutar, se recomienda cambiar en el "docker-compose.yml"la dirección IP de kafka de las variables "KAFKA_ADVERTISED_LISTENERS" "KAFKA_ZOOKEEPER_CONNECT", a la dirección IP del ordenador anfitrion donde se arrancará dichos servicios.

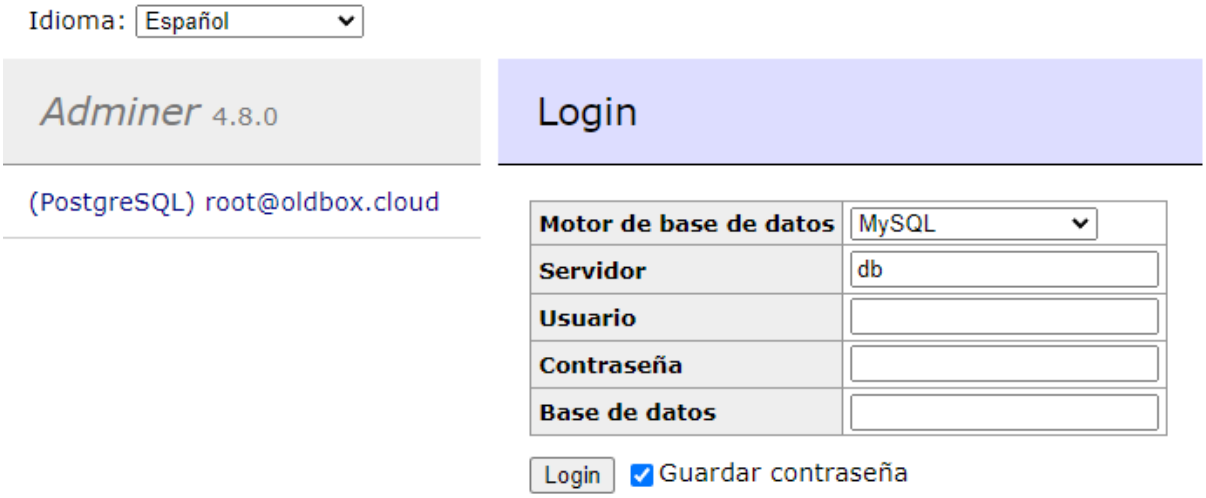
```
...
environment:
...
- KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://PonTuIPAquí:9092
...
- KAFKA_ZOOKEEPER_CONNECT=PonTuIPAquí:2181
...
```

Para arrancar la BD y Kafka, primero nos ubicaremos en la carpeta donde está alojado el archivo ´docker-compose.yml´, y ejecutaremos el comando:

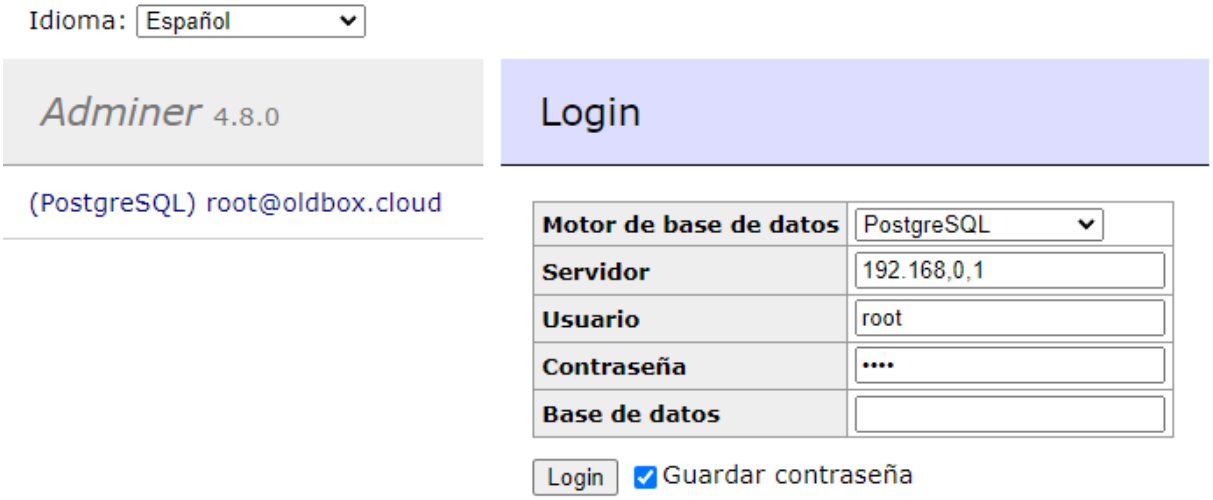
```
docker-compose up
```

Y dejamos que arranque del todo.

Una vez hecho esto, deberemos crear una base de datos en nuestro servidor de bases de datos, para ellos, una vez que nuestros contenedores de PostgreSQL, Adminer estén arrancados, abrimos a un navegador web cualquiera, y accederemos a ´localhost:8080´.

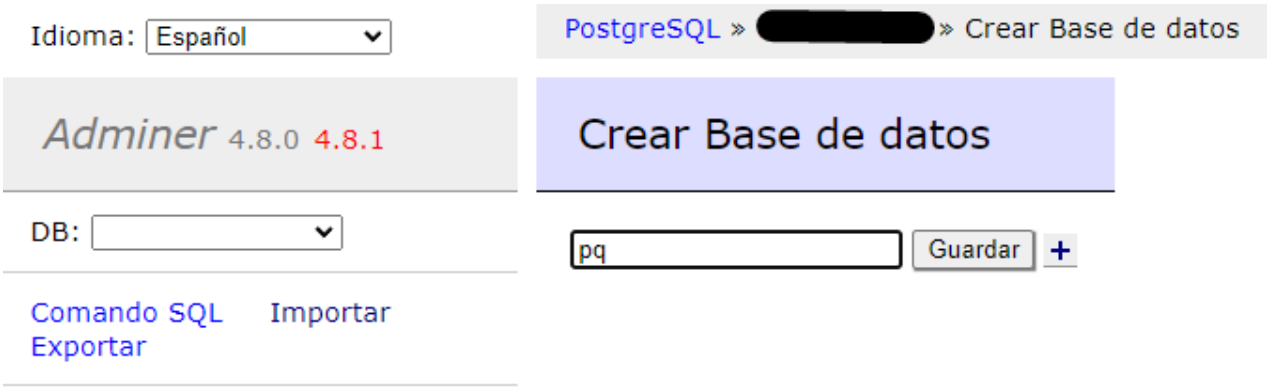


Una vez en el panel, seleccionaremos el Motor de bases de datos: PostgreSQL, Servidor: tu ip, Usuario: root, Contraseña: root, Bases de datos: vacio.

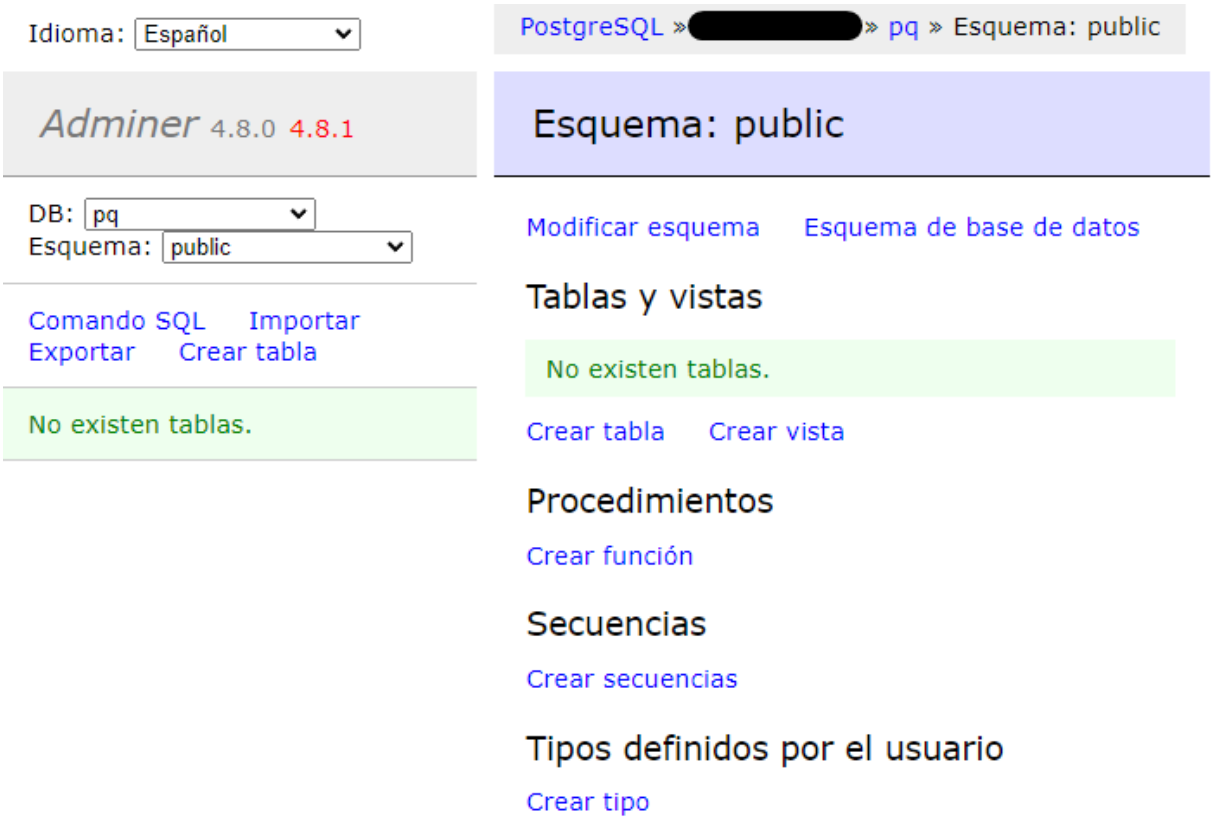


Una vez autenticados, pincharemos encima del botón 'crear base de datos'.

Le damos un nombre...



Y nuestra base de datos ya estaria creada.



Recuerda especificar el mismo nombre de base de datos al desplegar el Engine y Registry!

Ahora, una vez que tenemos la base de datos creada, y kafka arrancado correctamente, podemos pasar al despliegue de los módulos.

Para desplegar cada módulo, se necesitará la imagen encapsulada de cada módulo de docker.

3. Guía de Despliegue

Todos los módulos están encapsulados en imágnes de docker, cosa que nos permite crear uno o varios contenedores de docker con dichas imágenes, esto nos permite una alta adaptabilidad al sistema en el que se despliega, ya que no necesitamos depender de otros paquetes previos instalados en el sistema anfritrión, además de aislar cada módulo entre el resto del servicios del sistema, y incluso si lo queremos, configurarlo de tal manera que escale de manera automática, arrancando más engines, registries a medida que los usuarios usan la aplicación. Aún así, en esta guia nos limitaremos a explicar a como desplegar de manera sencilla el sistema.

3.1. Si tenemos las imágenes de antemano en un .tar

(Este método es el que se usará para desplegar en clase, ya que no queremos tardar 20 minutos solo para generar las imágenes.)

Antes de desplegar, importamos las imágenes en nuestro docker.

```
docker load -i imagenes.tar
```

De esta manera, ya tendríamos las imágenes cargadas en nuestro sistema, en caso de no poseer las imágenes pero si el código, sigue al siguiente apartado.

En la entrega no entregaremos las imágenes, ya que cada una llega a pesar entre 100MB y 900MB, ya que las imágenes no solo contienen cada módulo, sino también el SO y derivados, así que para la corrección (si se quiere probar) se deberán generar las imágenes correspondientes a cada módulo antes del lanzamiento de los contenedores.

3.2. No tenemos las imágenes, tenemos que generarlas

(Este es el método que se debe usar si se quiere arrancar el sistema, pero lo único que se tiene es el código fuente.)

Si no poseemos las imágenes de antemano a importar, las podemos generar para nuestro sistema usando el código fuente proveído.

Para esto, dentro de cada carpeta de los módulos, ejecutaremos el comando:

```
docker build -t sd:fwq_nombre .
```

De este modo, generamos las imágenes de cada módulo, donde “nombre” será el nombre del módulo. Por ejemplo, para el sensor, accederemos a la carpeta del código fuente del sensor.

```
cd fwq_sensor/
```

Y después, dentro de la carpeta del sensor, generamos la imagen con su nombre.

```
docker build -t sd:fwq_sensor .
```

Una vez generadas las imágenes de cada módulo, podremos arrancar los contenedores con sus debidos parámetros, como se muestra en el apartado siguiente.

3.3. Engine

Para ejecutar el modulo Engine, ejecutaremos el siguiente comando con los siguientes parámetros:

```
docker run --name fwq_engine -it \
-e AFORO=""  \
-e SECRET="" \
-e WTSADDRESS="" \
-e DBNAME="" \
-e DBUSER="" \
-e DBPASS="" \
-e DBADDR="" \
sd:fwq_engine
```

Siendo las palabras en mayúscula, los parámetros a definir para la aplicación a ejecutar.

- AFORO: el número de aforo disponible para el parque.
- WTSADDRESS: la dirección del servidor de sockets del Waiting Time Server.
- DBNAME: Nombre de la base de datos.
- DBUSER: Nombre del usuario de la base de datos.
- DBPASS: Contraseña del usuario de la base de datos.
- DBADDR: Dirección de la base de datos.
- SECRET: la palabra secreta que se usará para encriptar la transferencia en sockets, tiene que ser la misma para el Waiting Time Server.

Ejemplo:

```
docker run --name fwq_engine -it \
-e AFORO=15 \
-e SECRET=ABRACADABRA \
-e WTSADDRESS=http://172.20.40.151:9222 \
-e DBNAME=pq \
-e DBUSER=root \
-e DBPASS=root \
-e DBADDR=172.20.40.151 \
sd:fwq_engine
```

3.4. Registry

Para ejecutar el módulo Registry, ejecutaremos el siguiente comando con los siguientes parámetros, y PE:PI los puertos (Puerto Anfitrión : Puerto Contenedor) a exponer desde el contenedor docker a la máquina anfitriona:

```
docker run --name fwq_registry -it \
-e FWQRPORT="" \
-e DBNAME="" \
-e DBUSER="" \
-e DBPASS="" \
-e DBADDR="" \
-e SECRET="" \
-p PE:PI \
sd:fwq_registry
```

Siendo las palabras en mayúscula, los parámetros a definir para la aplicación a ejecutar.

- FWQRPORT: Puerto a escuchar el servidor de sockets del Registry.
- DBNAME: Nombre de la base de datos.
- DBUSER: Nombre del usuario de la base de datos.
- DBPASS: Contraseña del usuario de la base de datos.
- DBADDR: Dirección de la base de datos.
- SECRET: la palabra secreta que se usará para encriptar la transferencia en sockets, tiene que ser la misma para el Visitor.
- PE:PI: el número del puerto a escuchar, el mismo valor que especificamos en FWQPORT, ya que este parámetro se especifica qué puertos se exponen en el contenedor, y se enlazan a los puertos del sistema anfitrión.

Ejemplo:

```
docker run --name fwq_registry -it \
-e FWQRPORT=9090  \
-e DBNAME=pq \
-e DBUSER=root \
-e DBPASS=root \
-e DBADDR=172.20.40.151 \
-e SECRET=LOREIPSUM \
-p 9090:9090 \
sd:fwq_registry
```

3.5. Sensor

Para ejecutar el módulo Sensor, ejecutaremos el siguiente comando con los siguientes parámetros:

```
docker run --name fwq_sensor -it \
-e IDATTR="" \
-e X="" \
-e Y="" \
-e KAFKAADRDRESS="" \
-e IMAGEN="" \
-e ATTRPERSONTIMEINTER="" \
sd:fwq_sensor
```

Siendo las palabras en mayúscula, los parámetros a definir para la aplicación a ejecutar.

NOTA: Recuerda no poner 2 sensores con la misma id.

- IDATTR: ID de la atraccion (empezar desde 0)
- X: Coordenada X de la atraccion
- Y: Coordenada Y de la atraccion
- KAFKAADDRESS: Direccion del servidor Kafka.
- IMAGEN (OPTATIVO): Imagen a asociar a la atraccion para que se muestre en el mapa del visitante.
- ATTRPERSONTIMEINTER: Tiempo de espera para emitir las personas en cola de la atracción vía kafka.

Ejemplo:

```
docker run --name fwq_sensor -it \
-e IDATTR=0 \
-e X=2 \
-e Y=14 \
-e KAFKAADRDRESS=172.20.40.151:9092 \
-e IMAGEN=https://i.imgur.com/Ff7SEP6.png \
-e ATTRPERSONTIMEINTER=3 \
sd:fwq_sensor
```

3.6. Visitor

Para ejecutar el módulo Visitor, ejecutaremos el siguiente comando con los siguientes parámetros, y PE:PI los puertos (Puerto Anfitrión : Puerto Contenedor) a exponer desde el contenedor docker a la máquina anfitriona:

```
docker run --name fwq_visitor -it \
-e REACT_APP_REGISTRYADDRESS="" \
-e REACT_APP_KAFKACONTROLLER="" \
-e REACT_APP_KAFKAADDRESS="" \
-e REACT_APP_SECRET="" \
-e REACT_APP_VISITORINTERVAL="" \
-e REACT_APP_SENSORCHECKINTERVAL="" \
-e PE:5000 \
-e 9111:9111 \
sd:fwq_visitor
```

Siendo las palabras en mayúscula, los parámetros a definir para la aplicación a ejecutar.

NOTA: Para acceder al visitor recuerda abrir en el navegador la dirección del ordenador que hostea el visitor seguido de :5000. Ej: 172.20.40.151:5000

- REACT_APP_REGISTRYADDRESS: Dirección del servidor de sockets de Registry.
- REACT_APP_KAFKACONTROLLER: Dirección de la máquina anfitriona, para que los clientes (navegadores) puedan conectarse al controlador de Kafka.
- REACT_APP_KAFKAADDRESS: Dirección del servidor de Kafka.
- REACT_APP_SECRET: la palabra secreta que se usará para encriptar la transferencia en sockets, tiene que ser la misma para el Registry.
- PE:5000: Puerto al que enlazar el servidor web. En el contenedor, nuestro servidor web escucha en el puerto 5000, siendo PE el puerto que nosotros queremos que docker enlace a nuestro sistema anfitrión. Por ejemplo en 80.
- REACT_APP_VISITORINTERVAL: Tiempo de espera para que el visitante de un paso.
- REACT_APP_SENSORCHECKINTERVAL: Numero de iteraciones que tienen que suceder sin que una atraccion se actualice para que el visitante se de cuenta de que el sensor esta offline. (Tiene que ser por lo menos x2 mayor que VISITORINTERVAL).

Ejemplo:

```
docker run --name fwq_visitor -it \
-e REACT_APP_REGISTRYADDRESS=http://172.20.40.151:9090 \
-e REACT_APP_KAFKACONTROLLER=http://172.20.40.150:9111 \
-e REACT_APP_KAFKAADDRESS=172.20.40.151:9092 \
-e REACT_APP_SECRET=LOREIPSUM \
-e REACT_APP_VISITORINTERVAL=0.75 \
-e REACT_APP_SENSORCHECKINTERVAL=5 \
-p 80:5000 \
-p 9111:9111 \
sd:fwq_visitor
```

3.7. Waiting Time Server

Para ejecutar el módulo Waiting Time Server, ejecutaremos el siguiente comando con los siguientes parámetros, y PE:PI los puertos (Puerto Anfitrión : Puerto Contenedor) a exponer desde el contenedor docker a la máquina anfitriona:

```
docker run --name fwq_waitingtimeserver -it \
-e WTSPORT="" \
-e SECRET="" \
-e KAFKAADDRESS="" \
-p PE:PI \
sd:fwq_waitingtimeserver
```

Siendo las palabras en mayúscula, los parámetros a definir para la aplicación a ejecutar.

- WTSPORT: Puerto en el que escuchará el servidor de sockets de Waiting Time Server.
- SECRET: la palabra secreta que se usará para encriptar la transferencia en sockets, tiene que ser la misma para el Engine.
- KAFKAADDRESS: Dirección del servidor de Kafka.
- PE:PI: Puertos a exponer y mapear para servir el servidor de sockets.

Ejemplo:

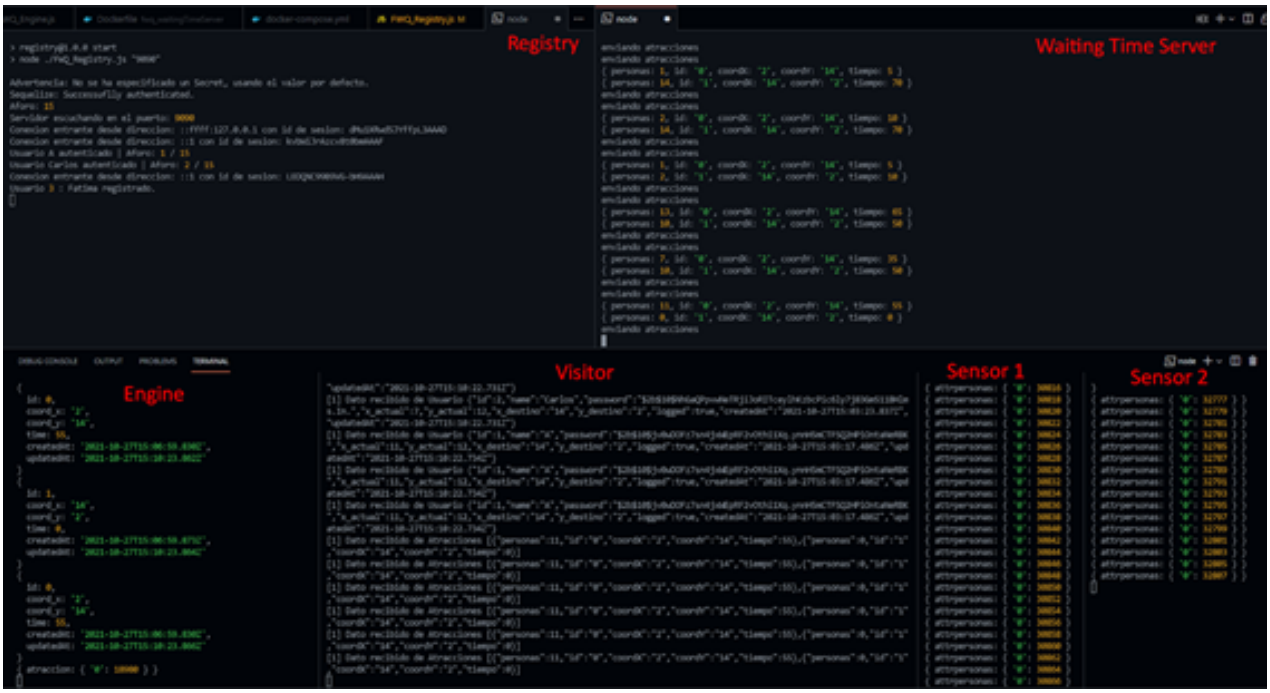
```
docker run --name fwq_waitingtimeserver -it \
-e WTSPORT=9222 \
-e SECRET=ABRACADABRA \
-e KAFKAADDRESS=172.20.40.151:9092 \
-p 9222:9222 \
sd:fwq_waitingtimeserver
```

3.8. Orden de lanzamiento

No hay un orden obligatorio de lanzamiento, se pueden arrancar los módulos en el orden que se quiera.

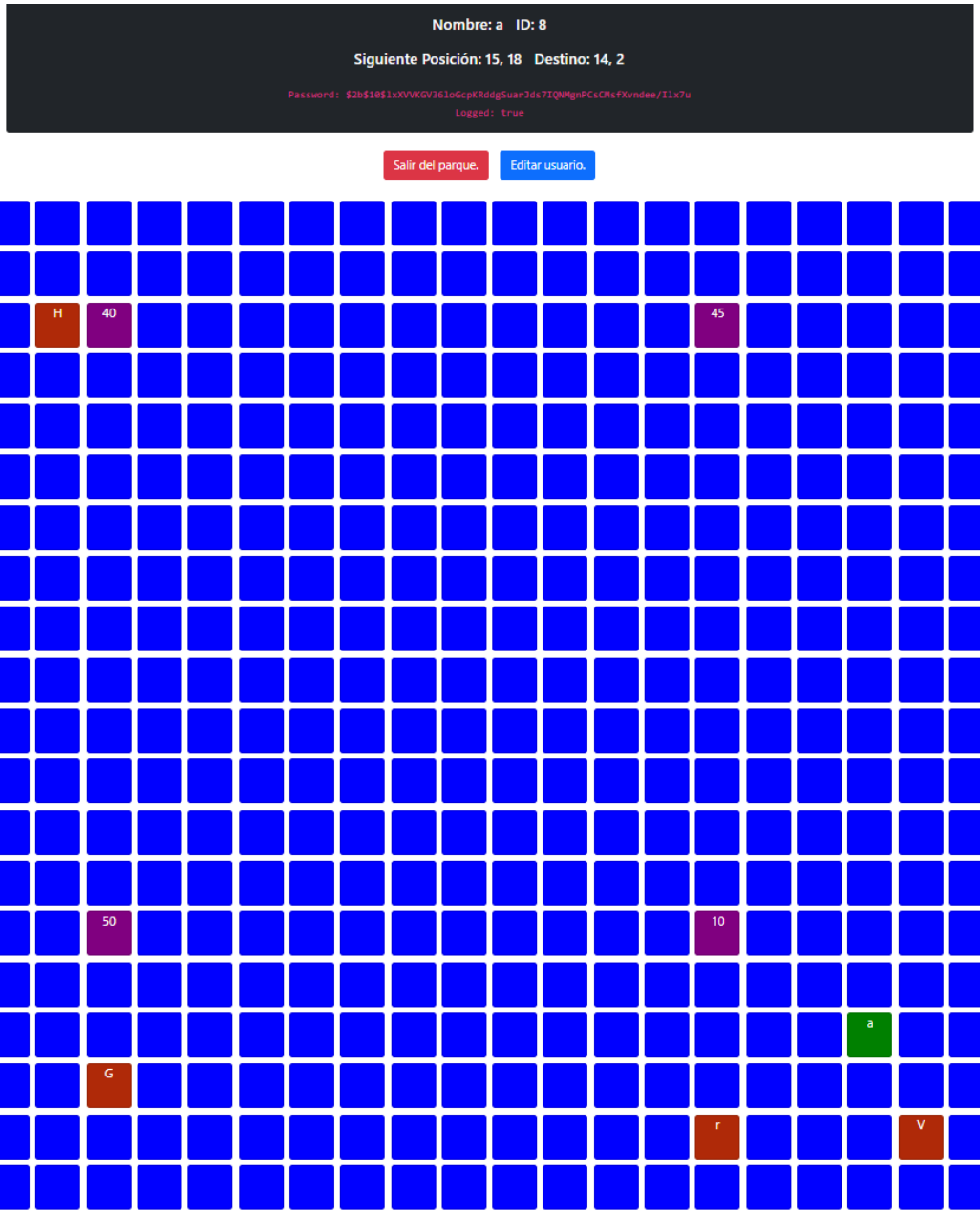
3.9. Capturas de funcionamiento

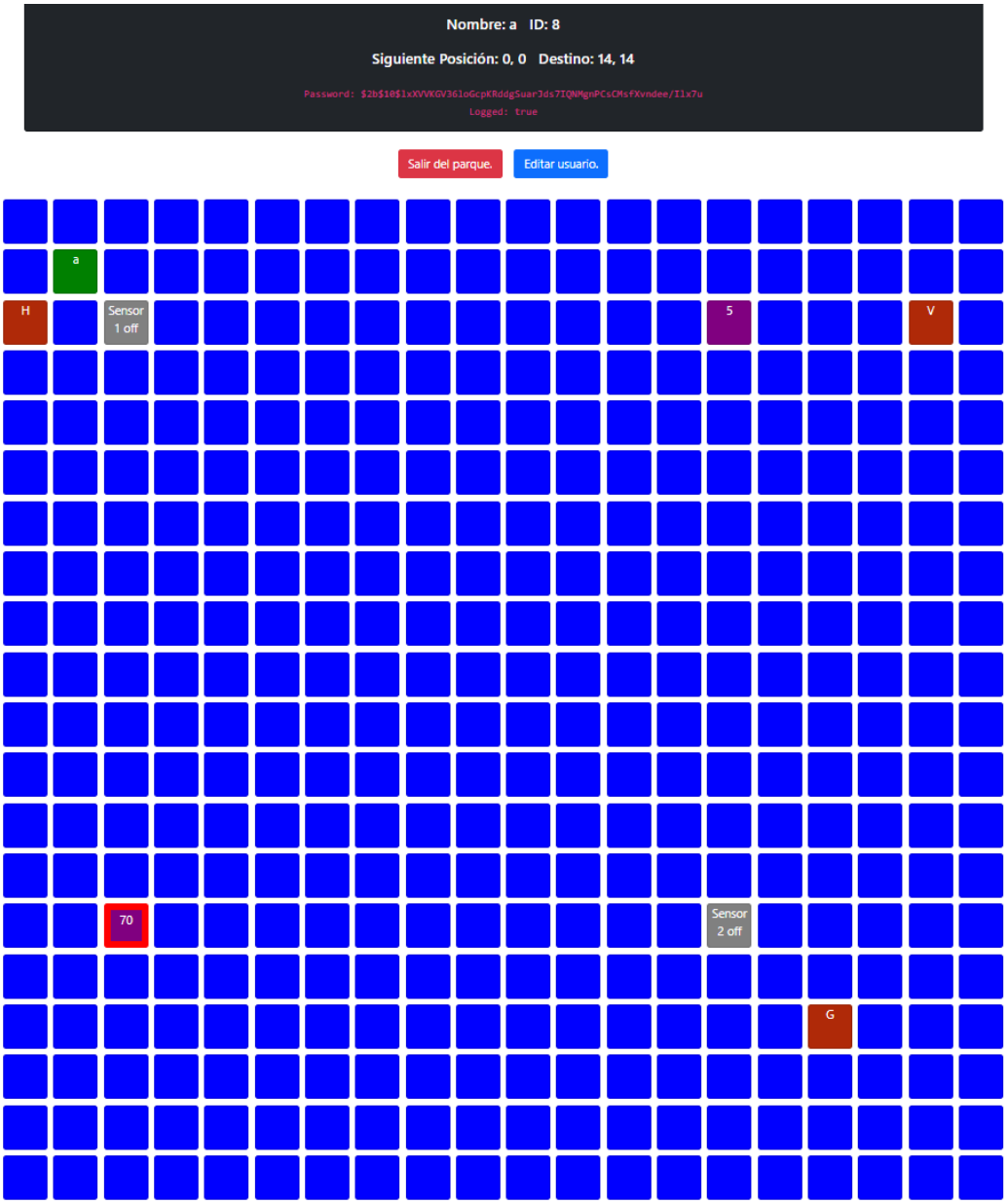
3.9.1. Consolas:



3.9.2. Visitante (Mapa):

- **Verde**: Visitante actual.
- **Naranja**: Otros vistantes.
- **Morado**: Atracciones con cola menor a 60 minutos.
- **Morado con borde rojo**: Atracciones donde el tiempo de cola es 60 o superior.
- **Gris**: Atracción cuyo sensor ha dejado de enviar datos (Sensor, Waiting time o Engine offline).


Nombre: a ID: 8
Siguiente Posición: 15, 18 Destino: 14, 2
Password: $2b$10$lxXVVKGV36loGcpKRddgSuarJds7IQNMgnPCsCMsfXvndee/Il x7u
Logged: true
Salir del parque.
Editar usuario.
H
40
45
50
10
a
G
r
V

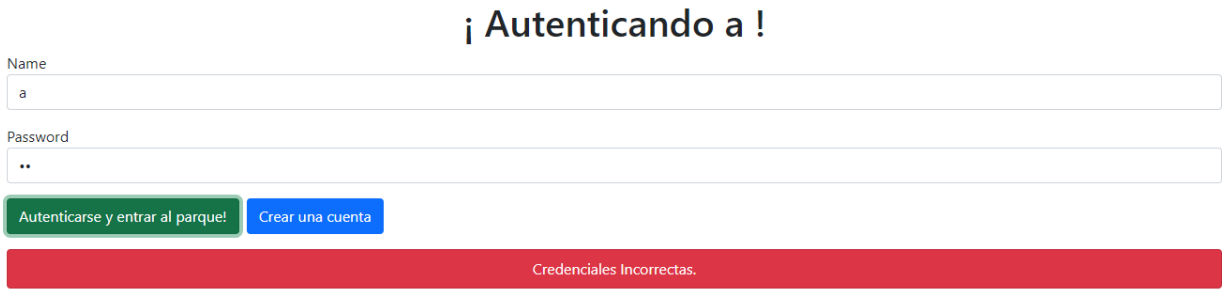


Leyenda colores visitor:

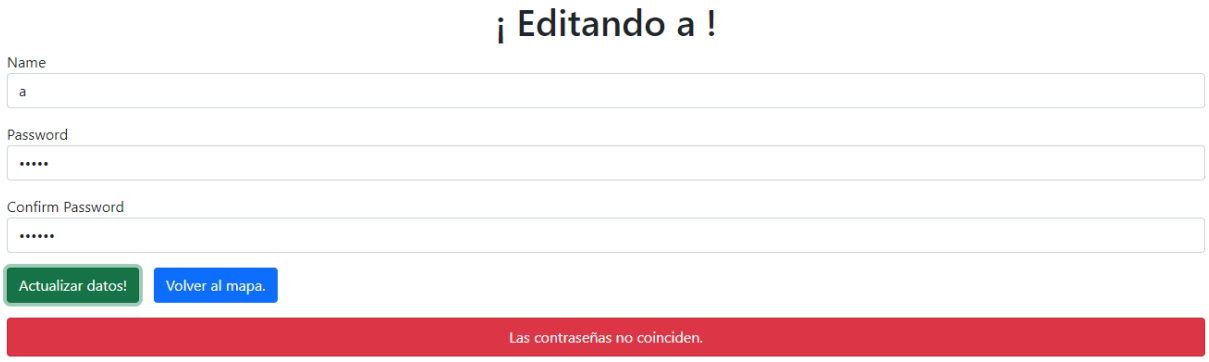
3.9.3. Visitante (Registro con usuario ya existente):

3.9.4. Visitante (Registro con contraseñas diferentes):

3.9.5. Visitante (login con credenciales erroneas):



3.9.6. Visitante (Ediatando cuenta con contraseñas diferentes):



3.9.7. Visitante (Registry o Visitor Offline):