

A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from the bar, containing the date.

19-6-2022

FUN WITH QUEUES

Memoria de la práctica

Several thin, curved lines in dark blue and light grey originate from the bottom left and sweep upwards and to the right.

Miguel Pérez Giménez – 74395666G
Carles Saez Marti – 35593837A

ÍNDICE

1. Explicación detallada de los elementos SW:

1.1: FWQ_REGISTRY

1.2: FWQ_VISITOR

1.3: FWQ_ENGINE

1.4: FWQ_WAITING_TIME_SERVER

1.5: FWQ_SENSOR

1.6: FRONT

1.1 FWQ Registry:

El Registry cumple con las siguientes funcionalidades:

- Registrar un visitante nuevo.
- Permitir al visitante editar su propio usuario.
- Comunicarse con el visitante mediante GRPC seguro.
- Comunicarse con el visitante mediante API.

Registrar visitante:

```
def registra(name, password):  
  
    conn = sqlite3.connect('../database.db')  
    cur = conn.cursor()  
  
    try:  
        pass_hash = hashlib.md5(bytes(password, encoding="utf8")).hexdigest()  
  
        cur.execute('insert into visitor(name, password) values("' + name + '", "' + pass_hash + '")')  
  
        conn.commit()  
  
        cur.execute('select id from visitor where name = "' + name + '"')  
        id_vis = cur.fetchall()  
        id_vis = id_vis[0][0]  
  
        conn.close()  
        return True, [str(id_vis), name]  
    except:  
        return False, ["ERROR al añadir a " + name + " al registro."]
```

Imagen1: función registra declarada en FWQ_Registry/controlloer.py.

Como vemos en la imagen, primero nos conectamos a la base de datos. A continuación, creamos un cursor que será el encargado de insertar el nuevo visitante haciendo uso de la funcionalidad execute. Para enviarle la información al usuario hemos decidido usar una tupla formada por un booleano y una tupla con la id del visitante junto con el nombre.

También cabe recalcar que hemos hecho uso de la librería hashlib la cual realiza el cifrado de la contraseña usando el tipo de cifrado md5. También hemos realizado las funciones que tienen relación con la base de datos en un fichero ajeno al main.py para así asegurarnos de que la aplicación sea más fácil de mantener en un futuro.

Permitir al visitante editar su propio usuario:

```
def edita(name, password, newName, newPassword):
    conn = sqlite3.connect('../database.db')
    cur = conn.cursor()

    #el nuevo nombre ya existe
    cur.execute('select * from visitor where name = "' + newName + '"')
    user = cur.fetchall()
    if len(user) >= 1:
        conn.close()
        return False, ["ERROR al editar visitante (nuevo nombre ocupado)."]

    #el usuario a editar no existe
    cur.execute('select * from visitor where name = "' + name + '"')
    user = cur.fetchall()
    if len(user) <= 0:
        conn.close()
        return False, ["ERROR al editar visitante (credenciales incorrectas)."]

    pass_hash = hashlib.md5(bytes(password, encoding="utf8")).hexdigest()
    new_pass_hash = hashlib.md5(bytes(newPassword, encoding="utf8")).hexdigest()

    try:
        cur.execute('update visitor set name = "' + newName + '", password = "' + new_pass_hash + '" where name = "' + name + '" and password = "' + pass_hash + '"')
        conn.commit()
        conn.close()
    except:
        return False, ["ERROR al editar usuario. (excepcion)"]

    return True, [str(user[0][0]), newName, 'Visitante editado correctamente.']
```

Imagen2: función edita declarada en FWQ_Registry/controlloer.py.

Para editar el usuario lo que hemos hecho ha sido pedir que el visitante le mande al Registry el nuevo nombre y la nueva contraseña.

La función edita comprueba que el usuario exista en la BBDD y si el usuario pertenece a la BBDD entonces se cambiará por el nuevo nombre y la nueva contraseña que ha elegido.

Hemos usado la librería hashlib para el cifrado de tipo md5 para la contraseña y también hemos tenido en cuenta los distintos tipos de errores con sus correspondientes mensajes de error.

Comunicarse con el visitante mediante GRPC seguro:

```
global REGISTRY_GRPC_PORT
global REGISTRY_GRPC_IP

server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
todo_pb2_grpc.add_TodoServicer_to_server(TodoServicer(), server)

ca_cert = open("client-cert.pem", 'rb').read()
private_key = open("server-key.pem", 'rb').read()
certificate_chain = open("server-cert.pem", 'rb').read()

credentials = grpc.ssl_server_credentials([
    [(private_key, certificate_chain)],
    root_certificates=ca_cert,
    require_client_auth=True
])

server.add_secure_port('[:,:]' + REGISTRY_GRPC_PORT, credentials)
server.start()

print('Starting server. Listening on port ' + REGISTRY_GRPC_PORT)

try:
    while True:
        time.sleep(86400)
except KeyboardInterrupt:
    os.system("ps aux | grep appReg.py | awk '{print $2}' | xargs -I{} kill -9 {} 2>/dev/null")
    server.stop(1)
```

Imagen3: main declarado en FWQ_Registry/main.py

En la imagen podemos ver como se ha realizado la conexión GRPC segura con el visitante.

Lo primero que hemos hecho ha sido crear los correspondientes certificados SSL mediante los comandos que proporciona OpenSSL, una vez se han generado los certificados hay que añadirseles a la variable “credentials”.

A continuación, creamos el servidor con el código que mostramos aquí abajo y ya tendríamos el servidor con conexión segura.

```
server.add_secure_port('[:,:]' + REGISTRY_GRPC_PORT, credentials)
server.start()
```

Imagen4: creación del server con GRPC seguro en FWQ_Registry/main.py

Para concluir, me gustaría añadir también que el servidor se mantiene a la escucha con un (while True:) y si ocurre alguna excepción de tipo “KeyboardInterrupt”, por ejemplo, cuando se cierra la terminal con control + C, entonces el servidor se parará.

Comunicarse con el visitante mediante API:

```
#get all users
@app.route('/user',methods = ['GET'])
def getAllUsers():
    try:
        success,data=seleccionaTodos()
    except:
        return jsonify({'ok': False,'msg' : 'Error en el acceso a la base de datos'}), 400

    if success:
        return jsonify(data[0]), 200
    else:
        return jsonify({'ok': False,'msg' : 'Error al seleccionar usuarios'}), 400

#get user
@app.route('/user/<name>', methods=['GET'])
def getUser(name):
    print(name)
    try:
        success,data=seleccionaUser('name')
    except:
        return jsonify({'ok': False,'msg' : 'Error en el acceso a la base de datos'}), 400

    if success:
        return jsonify(data[0]), 200
    else:
        return jsonify({'ok': False,'msg' : 'Error al crear el usuario'}), 400
```

Imagen5: métodos GET de la API de Registry FWQ_Registry/appReg.py

Haciendo uso de la librería de Flask hemos realizado los métodos GET, POST, PUT y DELETE.

Todas las llamadas devuelven un json que consta de un booleano ('ok'), un mensaje ('msg') y un código de respuesta HTTP.

GET:

Hay dos tipos, una llamada GET que te devuelve todos los usuarios y una llamada que te devuelve el usuario con el nombre que se pasa como parámetro. Las funciones "seleccionaTodos()" y "seleccionaUser('name') son las encargadas de realizar la búsqueda en la BBDD.

POST, PUT y DELETE:

El resto de los métodos se ha implementado de forma similar a los de GET, a continuación, se muestra el código de cada uno de ellos.

Para concluir, cabe añadir que todas las llamadas están en la url /user

```
#create user
@app.route('/user',methods=['POST'])
def createUser():

    dt = datetime.now()
    try:
        success,data=registra(request.form.get('name'),request.form.get('password'))
    except:
        logIntoFile([' + dt.strftime('%d/%m/%Y, %H:%M:%S') + ', ERROR] '+request.remote_addr+' Error de registro de usuario via API')
        return jsonify({'ok': False,'msg' : 'Error al crear el usuario'}), 400

    if success:
        logIntoFile([' + dt.strftime('%d/%m/%Y, %H:%M:%S') + ', ALTA] '+request.remote_addr+' via API: ' + request.form.get('name') + '')
        return jsonify({'ok': True,'msg' : 'Usuario creado correctamente'}), 201
    else:
        logIntoFile([' + dt.strftime('%d/%m/%Y, %H:%M:%S') + ', ERROR] '+request.remote_addr+' Error de registro de usuario via API')
        return jsonify({'ok': False,'msg' : 'Error al crear el usuario'}), 400
```

Imagen6: método POST de la API de Registry FWQ_Registry/appReg.py.

```
#update user
@app.route('/user', methods=['PUT'])
def editUser():

    dt = datetime.now()
    try:
        success,data=edita(request.form.get('name'),request.form.get('password'),request.form.get('newName'),request.form.get('newPassword'))
    except:
        logIntoFile([' + dt.strftime('%d/%m/%Y, %H:%M:%S') + ', ERROR] '+request.remote_addr+' Error de modificacion de usuario via API')
        return jsonify({'ok': False,'msg' : 'Excepción al editar el usuario'}), 400

    if success:
        logIntoFile([' + dt.strftime('%d/%m/%Y, %H:%M:%S') + ', MODIFICACION] '+request.remote_addr+' via API: ' + request.form.get('name') +
        ' to ' + request.form.get('newName') + '')
        return jsonify({'ok': True,'msg' : 'Usuario editado correctamente'}), 202
    else:
        logIntoFile([' + dt.strftime('%d/%m/%Y, %H:%M:%S') + ', ERROR] '+request.remote_addr+' Error de modificacion de usuario via API')
        return jsonify({'ok': False,'msg' : 'Error al editar el usuario'}), 400
```

Imagen7: método PUT de la API de Registry FWQ_Registry/appReg.py.

```
#delete user
@app.route('/user',methods=['DELETE'])
def deleteUser():

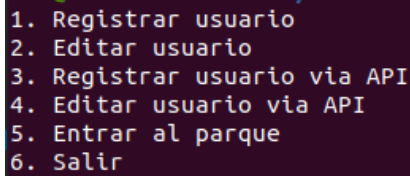
    dt = datetime.now()
    try:
        success,data=elimina(request.form.get('name'),request.form.get('password'))
    except:
        logIntoFile([' + dt.strftime('%d/%m/%Y, %H:%M:%S') + ', ERROR] '+request.remote_addr+' Error de eliminación de usuario via API')
        return jsonify({'ok': False,'msg' : 'Excepción al dar de baja el usuario'}), 400

    if success:
        logIntoFile([' + dt.strftime('%d/%m/%Y, %H:%M:%S') + ', BAJA] '+request.remote_addr+' via API: ' + request.form.get('name') + '')
        return jsonify({'ok': True,'msg' : 'Usuario dado de baja correctamente'}), 201
    else:
        logIntoFile([' + dt.strftime('%d/%m/%Y, %H:%M:%S') + ', ERROR] '+request.remote_addr+' Error de eliminación de usuario via API')
        return jsonify({'ok': False,'msg' : 'Error al dar de baja el usuario'}), 400
```

Imagen8: método DELETE de la API de Registry FWQ_Registry/appReg.py.

1.2 FWQ_VISITOR:

Al ejecutar el visitante se mostrará un menú con el siguiente aspecto:



```
1. Registrar usuario
2. Editar usuario
3. Registrar usuario via API
4. Editar usuario via API
5. Entrar al parque
6. Salir
```

Imagen9: menú principal FWQ_Visitor/main.py

El visitante hará uso de las funcionalidades que le proporciona “FWQ_Registry” al usar las opciones del 1 al 4. Cuando el visitante desea entrar al parque el “FWQ_Engine” será el encargado de manejar dicha acción, para ello el visitante tendrá que indicar sus credenciales y una vez el “FWQ_Engine” ha comprobado que coinciden con las de la BBDD ya le permitirá la entrada al parque.

Ahora se mostrará en la imagen de abajo como se ha realizado la comunicación con el “FWQ_ENGINE”, como podemos observar, se realizará con Apache Kafka y el visitante será tanto Consumer como Producer. Al recibir la respuesta del “FWQ_Engine” la función devolverá una tupla con un booleano (‘ok’), la posición en la que empieza (‘firstPos’) y un mensaje (‘msg’)



```
def parqueLogin(name, password):

    consumer = kc("loginTopic", bootstrap_servers = BROKER)
    parts = consumer.partitions_for_topic("loginTopic")

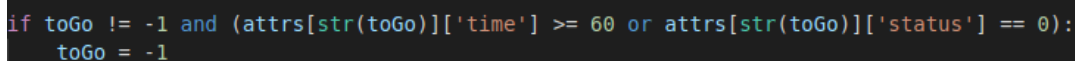
    prod = kp(bootstrap_servers=BROKER, value_serializer=lambda v: json.dumps(v).encode('utf-8'),acks='all')
    cons = kc("loginResponsesTopic", bootstrap_servers = BROKER)
    prod.send('loginTopic', {'name' : name, 'password' : password},partition=0)

    print("Esperando respuesta del login...")
    msg = {}
    msg = json.loads(next(cons).value.decode('utf-8'))

    if msg['ok']:
        firstPos = msg['firstPos']
        return msg['ok'], firstPos, msg['id_vis']
    else:
        return msg['ok'], None, None
```

Imagen10: parqueLogin() FWQ_Visitor/dentroParque.py

Y una vez esta dentro del parque se va moviendo hacia una atracción si el tiempo de espera y la temperatura son correctos.



```
if toGo != -1 and (attrs[str(toGo)]['time'] >= 60 or attrs[str(toGo)]['status'] == 0):
    toGo = -1
```

Imagen11: condición para decidir donde moverse FWQ_Visitor/movements.py

La función que se encarga de mover los visitantes es la función “entraAlParque”, la cual hace uso de TopicRecv y Topic para comunicarse con el Engine y con un topic personal que se crea con el nombre del visitante, una vez creado dicho topic el visitante se mueve según la función “moveAuto”.

```
def entraAlParque(name, firstPos):
    global LAST_ATTR

    producer = kp(bootstrap_servers=BROKER, value_serializer=lambda v: json.dumps(v).encode('utf-8'),acks='all')
    consumer = kc(name+'TopicRecv', bootstrap_servers = BROKER)

    producer.send(name+'Topic', {'ok':True, 'posX' : firstPos[0], 'posY' : firstPos[1]})
    time.sleep(1)

    toGo = -1
    pos = firstPos
    nextPos = pos
    toGoId = -1

    while True:

        producer.send(name+'Topic', {'ok':True, 'pos' : pos, 'next_pos': nextPos, 'to_go' : toGo })
        msg = json.loads(next(consumer).value.decode('utf-8'))

        pos = msg['new_pos']

        print(mapaToString(msg['mapa']))

        nextPos, toGo, LAST_ATTR, toGoId = moveAuto(msg['mapa'], pos, msg['attrs'], name, LAST_ATTR,toGoId)
        time.sleep(0.7)
```

Imagen12: entraAlParque() FWQ_Visitor/movements.py

```
def moveAuto(mapa, pos, attrs, name, lastAt, toGo):
    if lastAt != -1:
        mapa = mapa.copy()
        attrs = attrs.copy()
        try:
            attrs.pop(lastAt)
        except:
            pass

    inAttr = isInAttraction(name)
    if int(inAttr) != -1:
        goOut = timePassed(name, inAttr)
        if not goOut:
            return pos, pos, inAttr, toGo
        else:
            toGo = -1

    if toGo != -1 and (attrs[str(toGo)]['time'] >= 60 or attrs[str(toGo)]['status'] == 0):
        toGo = -1

    if toGo == -1 or toGo == None:
        _, toGo = getToGo(mapa, attrs, lastAt)

    else:
        toGo=searchAttrById(mapa,toGo)

    neight_occupied = neigh(mapa, pos, False)
    if toGo in neight_occupied:
        enterAttraction(name, mapa[toGo[0]][toGo[1]])
        return toGo, toGo, mapa[toGo[0]][toGo[1]], mapa[toGo[0]][toGo[1]]

    free = neigh(mapa, pos, True)

    newPos = pos

    xDist = toGo[0] - pos[0]
    if xDist > int(MAP_SIZE/2):
        newPos = xDown(pos)
```

Imagen13: moveAuto() FWQ_Visitor/movements.py

1.3 FWQ_ENGINE:

Es el encargado de manejar los distintos visitantes que aparecen en el parque, de recibir los tiempos de espera de las atracciones y la temperatura de cada región. También tiene una API para que el Front pueda usar la información que necesite en todo momento.

Comunicación GRPC Segura con el servidor de tiempos de espera:

El “FWQ_Engine” realiza la comunicación GRPC segura con el servidor de tiempos de espera de la misma forma que la realiza el “FWQ_Registry” con el visitante, es decir, generando los certificados OpenSSL correspondientes y haciendo uso de la función “grpc.add_secure_port” que nos proporciona la librería grpc.

```
def iniciarGrpcSecure():
    cert = open('client-cert2.pem', 'rb').read()
    key = open('client-key2.pem', 'rb').read()
    ca_cert = open('ca2.pem', 'rb').read()

    channel_creds = grpc.ssl_channel_credentials(ca_cert, key, cert)
    config = dotenv_values(".env")
    GRPC_WTS_IP = config['GRPC_WTS_IP']
    GRPC_WTS_PORT = config['GRPC_WTS_PORT']

    channel = grpc.secure_channel(GRPC_WTS_IP + ":" + GRPC_WTS_PORT, channel_creds)

    return todo_pb2_grpc.TODOStub(channel)
```

Imagen14: iniciarGrpcSecure() FWQ_Engine/grpc_funcs.py

Llamadas a la API OpenWeather:

Hemos creado un diccionario con 4 regiones y después hacemos la llamada a la API poniendo la ciudad que se pasa por parámetro y según el resultado de la temperatura ponemos el status a True si es correcta o False si no lo es.

```
def updateRegion(num):
    dic = {'Madrid': {'lat': '40.4167047', 'lon': '-3.7035825'}, 'Toronto': {'lat': '43.6534817', 'lon': '-79.3839347'}, 'Paris': {'lat': '48.8588897', 'lon': '2.3378371'}, 'London': {'lat': '51.5073219', 'lon': '0.1277582'}}
    ciudad = list(dic.keys())[num]

    response = requests.get('https://api.openweathermap.org/data/2.5/weather?lat=' + dic[ciudad]['lat'] + '&lon=' + dic[ciudad]['lon'] + '&appid=' + OPENWEATHER_API_KEY)
    temp = float(round(json.loads(response.text)['main']['temp'] - 273.15, 2))

    if temp < 20 or temp > 40:
        conn = sqlite3.connect('../database.db')
        cur = conn.cursor()
        cur.execute('update attraction set status = 0 WHERE region = "' + ciudad + '"')
        conn.commit()
        conn.close()
```

Imagen15: getWeather(ciudad) FWQ_Engine/appEng.py

Método GET para enviar el mapa al Front:

```
#get all users
@app.route('/map', methods = ['GET'])
def getMapApi():
    try:
        mapa, attrsDict, vis_arr = getMap()
    except:
        response = jsonify({'ok': False, 'msg' : 'Error en el acceso a la base de datos'})
        response.headers.add('Access-Control-Allow-Origin', '*')
        return response, 400

    response = jsonify({'mapa': mapa, 'attrs' : attrsDict, 'visitors': vis_arr})
    response.headers.add('Access-Control-Allow-Origin', '*')
    return response, 201
```

Imagen16: getMapApi() FWQ_Engine/map_funcs.py.

Método para cambiar el Status del visitante y para actualizar la posición:

```
def updatePosition(mapa, id_vis, pos, newPos):

    visStatus = None
    mapa = mapa.copy()
    if newPos != -1 and int(mapa[newPos[0]][newPos[1]]) < 0:
        return mapa, pos

    if newPos == -1:
        mapa = [[m if m != '-' + str(id_vis) else '0' for m in fila] for fila in mapa]
        saveMap(mapa, id_vis, 'disconnected')
        return

    for x in range(len(mapa)):
        for y in range(len(mapa[x])):
            if mapa[x][y] == "-" + str(id_vis):
                mapa[x][y] = '0'
            if x == newPos[0] and y == newPos[1]:
                if mapa[x][y] == '0':
                    visStatus = "walking"
                    mapa[x][y] = "-" + str(id_vis)
                else:
                    visStatus = str(mapa[x][y])

    saveMap(mapa, id_vis, visStatus)

    return mapa, newPos
```

Imagen17: updatePosition() FWQ_Engine/map_funcs.py.

1.4. FWQ WAITING TIME SERVER:

La función del servidor de tiempos de espera es calcular según los sensores el tiempo de espera que tiene una atracción. Para ello se conecta al Kafka como Consumer para obtener los datos de los sensores:

```
consumer = kc('SensorsTopic', bootstrap_servers = BROKER)
```

Imagen18: acceso al topic del sensor FWQ_WaitingTimeServer/main.py.

Para realizar el calculo del tiempo de espera se usa el numero de visitantes y el tiempo base de la atracción. El servidor manda la información al Engine cada segundo mediante Kafka.

```
attrs = {}
start = datetime.now()
end = start
while True:

    to_del = []
    for a in attrs:
        if attrs[a] == 0:
            to_del.append(a)
    for d in to_del:
        attrs.pop(d)

    msg = json.loads(next(consumer).value.decode('utf-8'))

    attr_id = str(msg['attr'])

    if attr_id in attrs.keys():
        attrs.pop(attr_id)

    if msg['people_count'] != 0:
        attrs[attr_id] = msg['people_count'] * int(TIME_PERSON) + int(TIME_BASE_ATTR)

    if end - start >= timedelta(seconds=1):
        print(str(attrs) + ' | (last message from attraction '+str(msg['attr'])+')')
        start = datetime.now()
    end = datetime.now()
```

Imagen19: main() FWQ_WaitingTimeServer/main.py.

1.5: FWQ SENSOR:

El sensor es un servidor que esta en un intervalo de entre 1 y 4 segundos enviando el numero de personas que hay en una atracción al servidor de tiempos de espera.

```
while True:
    try:
        with open('../FWQ_Sensor/fisic_attractions/attr'+str(sensor_id)+'.json', 'r') as a:
            attr_queue = json.load(a)
            names = list(attr_queue.keys())

            prod = kp(bootstrap_servers=BROKER, value_serializer=lambda v: json.dumps(v).encode('utf-8'),acks='all')
            prod.send('SensorsTopic', {'attr': sensor_id, 'people_count' : len(names)})

    except:
        pass

    timePassed = random.randrange(1,4)
    time.sleep(timePassed)
```

Imagen20: main() FWQ_Sensor/main.py.

1.6: Front:

Para la realización del Front hemos empleado el Framework **React.js**. Hemos incorporado estilos en el fichero App.css, como mostramos a continuación:

```
.mapDrawing {
  height: 23px;
  width: 23px;

  background-color: #313b4d;
}

.mapVisGroup{
  margin-top: 50px;
  display: flex
}

.mapTable {
  margin-left: auto;
  margin-right: auto;
}

.attractionsTable{
  margin-left: auto;
  margin-right: auto;
}

.visitorsTable{
  margin-left: auto;
  margin-right: auto;
}

.rightThing {
  margin-left: auto;
  margin-right: auto;
}

.App-header {
  min-height: 100vh;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
}
```

Imagen21: Estilos css frontend/src/App.css.

Hemos incorporado al App.js un componente y ahí es donde manejamos toda la lógica del front.

```
function App() {

  useEffect(() => {
  })

  return (
    <div className="App">
      <div>
        <ParkMap></ParkMap>
      </div>
    </div>
  );
}

export default App;
```

Imagen22: llamada a ParkMap declarado en frontend/src/App.js.

Si ejecutamos el Front podemos ver como se muestra el mapa, el estado de los visitantes, además se muestra si algún modulo falla y también toda la información de las atracciones. Cada región es un color diferente del mapa y los cuadrados verdes son visitantes y los negros atracciones.

The screenshot displays the application interface with three main sections:

- Left Panel (Visitors):** A table listing 23 visitors with their ID, Name, and Status.
- Center Panel (Map):** A 20x20 grid map where different colors represent regions. Black squares indicate attractions, and green squares indicate visitors. Numbers 1 through 20 are placed on the grid to identify specific locations.
- Right Panel (Attractions):** A table listing 20 attractions with their ID, Time, Region, Status, and a 'Check' button.

At the bottom of the interface, there are three buttons: 'WTS', 'Registry', and 'Engine'.

ID	NAME	STATUS
1	asd	disconnected
2	qwe	disconnected
3	zxc	disconnected
4	a	walking
5	b	4
6	c	walking
7	d	11
8	e	6
9	f	walking
10	g	10
11	h	walking
12	i	walking
13	j	walking
14	k	walking
15	l	walking
16	m	disconnected
17	n	disconnected
18	o	disconnected
19	p	disconnected
20	q	disconnected
21	r	disconnected
22	s	disconnected
23	t	disconnected

ID	TIME	REGION	STATUS	SENSORES
1	50	Madrid	[O]	OK Check
2	50	Toronto	[O]	OK Check
3	50	Toronto	[O]	OK Check
4	55	Madrid	[O]	OK Check
5	50	Toronto	[O]	OK Check
6	55	Madrid	[O]	OK Check
7	50	Madrid	[O]	OK Check
8	50	Madrid	[O]	OK Check
9	50	Toronto	[O]	OK Check
10	55	Madrid	[O]	OK Check
11	55	Toronto	[O]	OK Check
12	55	Madrid	[O]	OK Check
13	50	Madrid	[O]	OK Check
14	50	Toronto	[O]	OK Check
15	50	Oslo	[X]	OK Check
16	50	Oslo	[X]	OK Check
17	50	Paris	[O]	OK Check
18	50	Oslo	[X]	OK Check
19	50	Oslo	[X]	OK Check
20	50	Oslo	[X]	OK Check

Imagen23: ejecución con el frontEnd, con todo funcionando.

A continuación, se ha cerrado el Registry y como podemos ver se ha puesto rojo.

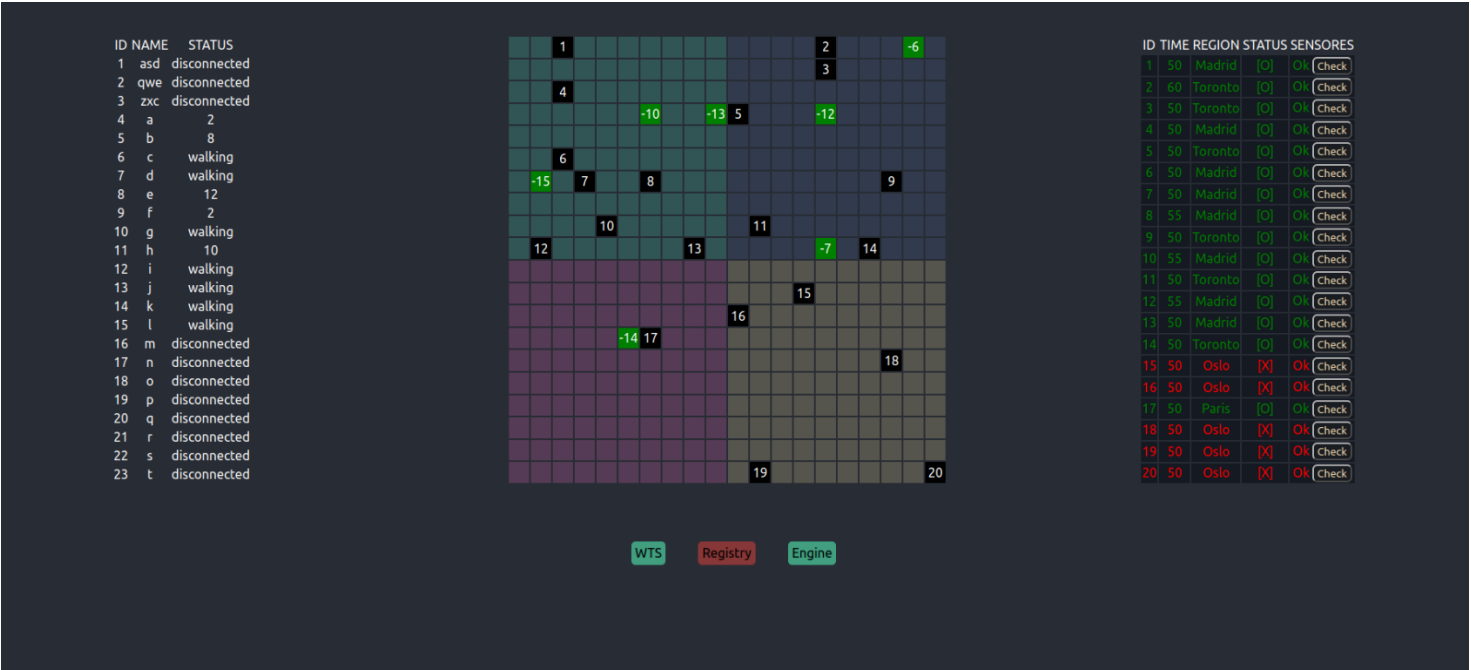


Imagen24: ejecución con el frontEnd, sin el Registry estando operativo.

Para concluir, por tema de tiempo no hemos podido hacer que se muestre cuando un sensor deja de funcionar. Si se aprieta el botón de “check” te devuelve una llamada a la API, pero te lo devuelve en formato json, ha faltado simplemente dejarlo bien estéticamente.