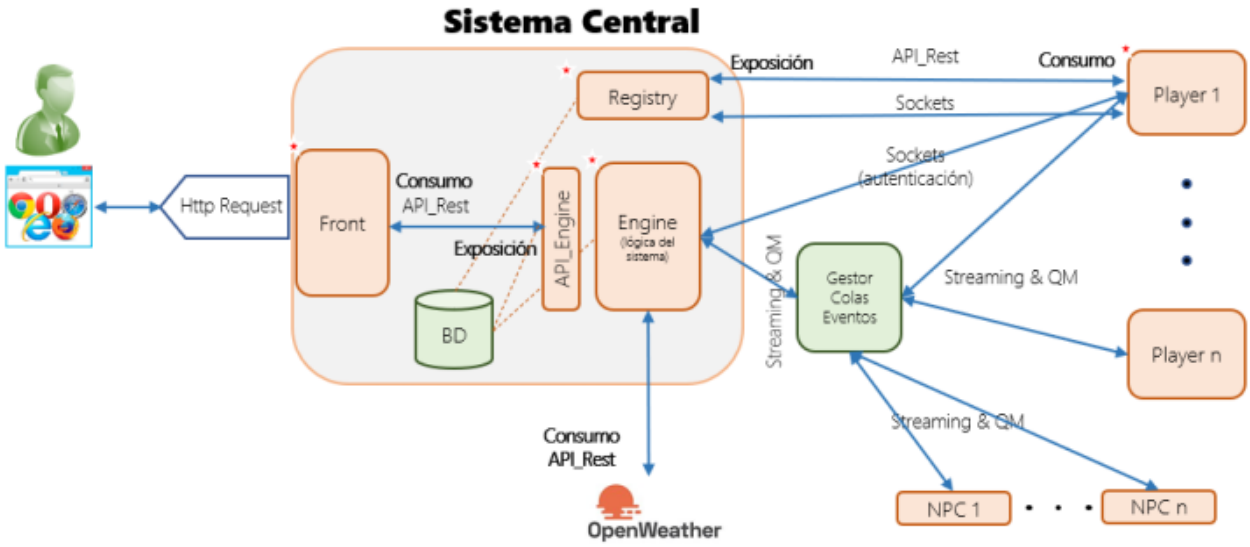
Sistemas distribuidos

Práctica no guiada: Seguridad y API Rest



Autores:
Vadym Formanyuk: X5561410X
Alicja Wiktoria Hyska: Y3739837M

Índice

Resumen

En el siguiente documento explicaremos a detalle la estructura y las partes funcionales del proyecto, cuyo objetivo es aplicar los sistema de seguridad y realizar el uso de API REST aplicable a los sistemas distribuidos.

Esta segunda parte de proyecto consta de partes como la aplicación de consumo de de API_rest, la implementación de seguridad con jugadores y la creación del Frontend, que permitirá visualizar la partida en el navegador.

El lenguaje de programación utilizado para la implementación de la práctica es *Python*, el entorno para la implementación de frontend es *Flask*.

1. Frontend

La parte de frontend consistirá en la construcción de una página web que muestre el mapa en curso y el estado de los jugadores.

Para realizar la parte del frontend se han utilizado Flask para desplegar las API, y SVELTE para la parte visual del mapa.

1.1. Flask

Para el servidor de front se ha usado Flask. La idea de Flask es poder acceder a los métodos que se han creado en el servidor Flask mediante API.

Flask es un framework minimalista escrito en Python que permite crear aplicaciones web rápidamente y con un mínimo número de líneas de código. Está basado en la especificación WSGI de Werkzeug y el motor de templates Jinja2 y tiene una licencia BSD.

Se han creado 2 archivos que serán ambos servidores con su ip y puerto. El primer servidor es 'API_Engine' , y el segundo 'API_ Registry':

- API_Engine. Este archivo contendrá únicamente 2 métodos de acceso mediante api GET. Un método para obtener los jugadores de la partida, y otro método para obtener el mapa.
- API_Registry. Este archivo contendrá únicamente 2 métodos de acceso mediante api PUT Y POST. El método de POST será para insertar un jugador en la base de datos, mientras que el método PUT para modificar los datos del jugador, que en este caso sólo será la contraseña como la única modificable de la práctica.

1.2. Vite + Svelte + bootstrap

Para el aspecto visual del frontend se ha usado vite,svelte y boostrap. Svelte es un compilador Front End gratuito y de código abierto creado por Rich Harris y mantenido por los miembros del equipo central de Svelte. Las aplicaciones Svelte no incluyen referencias al framework. En su lugar, la compilación de una aplicación Svelte genera código para manipular el DOM, lo que puede reducir el tamaño de los archivos transferidos, así como proporcionar un mejor arranque y rendimiento en tiempo de ejecución al cliente. Svelte tiene su propio compilador, que en el momento de la construcción convierte el código de la aplicación a JavaScript para el cliente. Está escrito en TypeScript. El código fuente de Svelte tiene licencia MIT y está alojado en GitHub.

Primero se arranca el servidor de 'API Engine' y luego el servidor del svelte mediante el siguiente comando:

- npm run dev – –host

Esto lo que hará es que el servidor de svelte vaya consultando cada segundo mediante API al servidor de 'API Engine' por el mapa y jugador en curso de la partida y lo mostrará en el navegador.

2. Cambio en Sistema Central

El núcleo del sistema, respecto al proyecto base, ha sido modificado en los siguientes aspectos:

- Registry (API_Registry): se aplicará la implementación de API_rest para el registro o la modificación y actualización de los jugadores.
- Engine (API_Engine): la lógica principal del juego ahora incluirá el consumo de API_rest, un resguardo actualizable del mapa en la base de datos y la implementación de seguridad con los jugadores.
- Database: a diferencia de la segunda práctica, la base de datos del sistema ahora condenrá el mapa y el estado de los jugadores en una partida.

2.1. API_Registry

Adentrándonos en la parte de Flask, hemos mencionado los dos nuevos servidores a los que daremos uso para la parte de API en esta tercera práctica. La implementación de flask en las API's va a permitir un uso visual y sencillo a la hora de la implementación.

```python
# EL POST ES PARA AÑADIR
@app.route('/jugador', methods= ['POST'])
@cross_origin()
def addJugador():

    aes_key_temporal = bytes(eval(request.json.split("|&|")[1]))
    datos = eval(api_desencriptar(aes_key_temporal , request.json.split("|&|")[0]))

    conexion_db = db()
    if(conexion_db.openCommunication(SERVIDOR_IP , USUARIO , CONTRASENYA, BASE_DE_DATOS) == True):

        if len(conexion_db.getJugador(datos['alias'])) > 0: # si el jugador con ese alias existe
            return f"{api_encriptar(aes_key_temporal,'ERROR: Este jugador ya existe.')}"
        else:
            if conexion_db.insertar_Jugador(datos['alias'], datos['password'], datos['nivel'], datos['ef'], datos['ec'], datos['posicion']):
                return f"{api_encriptar(aes_key_temporal,'Jugador creado.')}"
            else:
                return f"{api_encriptar(aes_key_temporal,'ERROR: Jugador no creado.')}"
    else:
        print("No se ha podido establecer comunicación con la base de datos en insertar jugador.")
        return f"{api_encriptar(aes_key_temporal,'Error no hay conexión con la DB.')}"

# EL PUT ES PARA MODIFICAR
@app.route('/jugador', methods = ['PUT'])
@cross_origin()
def editJugadorPassword():

        aes_key_temporal = bytes(eval(request.json.split("|&|")[1]))
        datos = eval(api_desencriptar(aes_key_temporal , request.json.split("|&|")[0]))

        conexion_db = db()
        if(conexion_db.openCommunication(SERVIDOR_IP , USUARIO , CONTRASENYA, BASE_DE_DATOS) == True):

            if len(conexion_db.getJugador(datos['alias'])) > 0: # si el jugador con ese alias existe

                if conexion_db.modJugador_Password(datos["alias"],datos["password"]):
                    return f"{api_encriptar(aes_key_temporal,'Jugador modificado.')}"

                else:
                    return f"{api_encriptar(aes_key_temporal,'Jugador no modificado.')}"
            else:
                return f"{api_encriptar(aes_key_temporal,'ERROR: jugador no existe.')}"
        else:
            print("No se ha podido establecer comunicación con la base de datos en modificar jugador.")
            return f"{api_encriptar(aes_key_temporal,'Error no hay conexión con la DB.')}"

```

En este caso, se ha creado una clase nueva que nos dará la opción de interactuar con el jugador, permitiendo el uso de API para registrarse o modificar uno de sus datos (en este caso, la contraseña, ya que los demás atributos se asignan de manera aleatoria). Y ya hemos mencionado anteriormente que cosa importante por implementar en referencia al API son los recursos.

Toda API REST se basa en acceder y/o manipular recursos. Para ello, las APIs utilizan el protocolo HTTP y alguno de sus verbos. Generalmente, los verbos más utilizados en un API REST son los siguientes:

- **GET**: Para obtener un recurso o colección.
- **POST**: Para crear un recurso (y/o añadirlo a una colección).
- **PUT**: Para modificar o actualizar un recurso.
- **DELETE**: Para eliminar un recurso.

Para implementar los recursos en Flask lo hacemos de manera que un recurso no es más que una clase asociada a un endpoint (la URL mediante la que se expone el recurso) que define cómo se puede acceder y/o manipular dicho recurso. Para ello, solo hay que implementar los métodos correspondientes a cada uno de los verbos HTTP que se necesiten.

Por ejemplo, así es como se vería API_Registry aplicdando un método POST, que ya sabemos que sirve para crear un recuso, que en este caso es, la dada de alta de un jugador (el registro de un jugador nuevo):

```
@app.route('/jugador', methods= ['POST'])
@cross_origin()
def addJugador():

    conexion_db = db()
    if(conexion_db.openCommunication(SERVIDOR_IP , USUARIO , CONTRASENYA, BASE_DE_DATOS) == True):

        if len(conexion_db.getJugador(request.json['alias'])) > 0: # si el jugador con ese alias existe
            return 'ERROR: Este jugador ya existe.'
        else:
            if conexion_db.insertar_Jugador(request.json['alias'], request.json['password'], request.json['nivel'], request.json['ef'], request.json['ec'], request.json['posicion']):
                return 'Jugador creado.'
            else:
                return 'ERROR: Jugador no creado.'

    else:
        print("No se ha podido establecer comunicación con la base de datos en insertar jugador.")
        return 'Error no conexión con la DB.'
```

Como podemos ver, nos conectamos a la base de datos y, a continuación, hacemos el uso de los métodos que ya han sido implementados anteriormente, por ejemplo, el método "getJugadores()" nos devolverá a los jugadores que en su campo "alias" contengan la cadena que se nos pasa por el parámetro. A continuación, usamos el método "insertar_Jugador()", que al pasarle todos los datos desde un json crea un jugador nuevo.

En total la clase tiene dos funciones, la de agregar un jugador y la de modificar la contraseña, que se haría de una manera muy similar a ésta, pero haciendo el uso del método [PUT].

2.1.1. Registro de auditoría

En esta parte, también se ha implementado el registro de auditoría, que en este caso, en un archivo aparte llamado "LOGS.txt" que nos muestra los datos que un evento sucedido en API_Registry.

El archivo de texto se crea con la instrucción:

open("nombre_fichero.txt", "a")

El tiempo exacto del momento en el que ocurre evento lo sacaremos de la librería **datetime** que se implementa para python, y para obtener el ip de la máquina que realiza el evento, haremos un **request.remote_addr**.

Se aplica de tal manera que cuando ocurre un evento, con la instucción *fichero.write()* se describe lo que quiere que contenga la línea, ya sea un mensaje de error o un evento de creación o modificación completada. A continuación, se cierra el fichero con *fichero.close()*.

Tomando como ejemplo un mensaje de error, aquí podemos ver que el jugador no se crea si no se consigue una comunicación entre una función de la base de datos:

```
if conexion_db.insertar_Jugador(datos['alias'], datos['password'], datos['nivel'], datos['ef'], datos['ec'], datos['posicion']):
    ##AUDITORIA
    fichero.write(str(date) + " | " + str(ip) + " | JUGADOR CREADO "+'\n')
    fichero.close()
    ###########
    return f"{api_encriptar(aes_key_temporal,'Jugador creado.')}"
else:
    ##AUDITORIA
    fichero.write(str(date) + " | " + str(ip) + " | ERROR | JUGADOR NO CREADO "+'\n')
    fichero.close()
    ###########
    return f"{api_encriptar(aes_key_temporal,'ERROR: Jugador no creado.')}"
```

Entonces una vez que eso ocurre, se escribirá en el fichero un mensaje de error, con la hora del mismo y el ip de la máquina que ha producido el error:

```
LOGS.txt
1   2022-12-18 20:04:32.049692 | 192.168.100.4 | ERROR | JUGADOR NO CREADO
```

2.2. API_Engine

```
@app.route("/mapa")
@cross_origin()
def mapaPartida():

    mapa = getMapa_en_BaseDatos()

    return jsonify(mapa)


def getJUGADORES_PARTIDA_ALIAS_en_BaseDatos() -> list:
    ##ABRO CONEXION CON LA BASE DE DATOS
    m = []
    conexion_db = db()
    if(conexion_db.openCommunication(SERVIDOR_IP , USUARIO , CONTRASENYA, BASE_DE_DATOS) == True):
        m = conexion_db.get_JUGADORES_PARTIDA_ALIAS()
        m = [tuple(l) for l in m] #convierto lista de listas en lista de tuplas (requerido en algunos métodos)
    else:
        print("ERROR: no se ha podido establecer comunicación con la DB en getJUGADORES_PARTIDA_ALIAS_en_BaseDatos()")

    return m

@app.route("/jugadores")
@cross_origin()
def jugadoresPartida():
    return jsonify(getJUGADORES_PARTIDA_ALIAS_en_BaseDatos())

```

Al igual que en la clase API_Registry(), usaremos los mismos recursos y los mismos métodos de implementación de API en flask, pero en este caso, en lugar de ir orientado a los jugadores, éste se ocupará de dar visibilidad a la partida, mostrado el mapa y los jugadores que interactúan en la misma partida.

Para el mapa, necesitamos crear un objeto que la contenga y simplemente devolvemos el objeto con jsonify. En el caso de los jugadores, recogerá únicamente a los jugadores que están en la partida, (tanto para ello, como para todo lo demás visto anteriormente para las conexiones con la base de datos, hay una función en la clase *db.py*).

3. Opciones del jugador

Para poder elegir el método, si api o sockets, de registro/modificación de usuario se han añadido 2 nuevas opciones en el menú:

```
option = input()
if option == '1':
    registrarUsuario("SOCKETS")
elif option == '2':
    registrarUsuario("API")
elif option == '3':
    editarPerfil("SOCKETS")
elif option == '4':
    editarPerfil("API")
```

En la práctica anterior únicamente pudimos crear o actualizar un jugador con la implementación de sockets, pero en este caso en la clase de player se han añadido dos métodos que en lugar de invocar directamente al registry, actúa dependiendo de la cadena que se le pase como parámetro, que puede ser "SOCKET" o "API".

Como ejemplo, aquí apreciamos la función modificada para registrar un jugador:

```
def registrarUsuario(tipo):
    print("REGISTRO DE USUARIO")
    op="N"
    alias = password = ""

    while op=="N":

        while True:
            print("Introduce tu alias:")
            alias=input()
            if(alias == ""):
                print("Tiene que introducir un alias.")
            else:
                break

        while True:
            print("Introduce tu contraseña :")
            password=input()
            if(password == ""):
                print("La contraseña no puede estar vacía. Si desea salirse, introduzca una letra y ponga 'n' en la confirmación")
            else:
                break

        print("Compruebe sus datos: ")
        print("Alias: "+str(alias)+" Contraseña: "+str(password))
        op="A"
        while op!="N" and op!="S" and op!="n" and op!="s":
            print("S/N")
            op=input()
            if op=="S" or op=="s":

                posX = random.randrange(0,19)
                posY = random.randrange(0,19)
                ef_ = random.randrange(-10,10)
                ec_ = random.randrange(-10,10)
                pos = str(posX) + "," + str(posY)
                mensaje_crear_perfil = {f"alias":alias, "password":password,"nivel":1,"ef":ef_,"ec":ec_,"posicion":pos}
                json_mensaje = json.dumps(mensaje_crear_perfil)

                #Creo una llave temporal de 1 solo uso para el socket entre player a registry (PRÁCTICA 3)
                salt = b'\x1a\xb3m\xa0\xbf!\xfc\xac6\x7f"\x9f}\xc3\xe0M\xe7\xc4\x94\xc6s\xfc\xa8\x08\xa6C\x19\x9d\x90\xb6\xec\xf3'
                password="vadim" #un password elegido (necesario para crear la key)
                aes_key_temporal = PBKDF2(password,salt,dkLen=32)

                if tipo == "SOCKETS": print(registrarUsuario_sockets(json_mensaje, aes_key_temporal))
                else:
                    if tipo == "API": print(registrarUsuario_API(json_mensaje, aes_key_temporal))
```

Por lo que ahora, cuando ejecutemos la clase del player en la terminal, se nos mostrará de la siguiente manera:

```
vadym@vadym-VirtualBox:~/Escritorio/SD P3$ python3 AA_Player.py
¡BIENVENIDO AL JUEGO!
Seleccione una opcion:
1. Crear perfil (SOCKETS)
2. Crear perfil (API)
3. Editar un perfil existente (SOCKETS)
4. Editar un perfil existente (API)
5. Unirse a la partida
6. Salir
```

4. Servidor de clima: Openweather

La página a que se conecta mediante la api es OpenWeatherMap. Dicha página es un servicio en línea que proporciona datos meteorológicos globales a través de API, incluidos datos meteorológicos actuales, pronósticos, predicciones inmediatas y datos meteorológicos históricos para cualquier ubicación geográfica.

Usando la API key creada en la página de opeweather se saca la temperatura mediante las siguientes lineas de código:

args.json

```
"AA_Engine":[
  {
  "puerto_escucha":8202,
  "max_jugadores":4,
  "ip": "localhost",
  "consumidor_timeout":160000,
  "API_KEY_OPENWEATHER": "62994c42dc83d47d88e26d201585ec80",
  "CIUDADES": ["Tokio","Sidney","Paris","Pekin"]
  }
```

AA_Engine.py

```
API_KEY_OPENWEATHER =  args_json['AA_Engine'][0]['API_KEY_OPENWEATHER']
ciudades : list = args_json['AA_Engine'][0]['CIUDADES']

try:
    for i, ciudad in enumerate(ciudades):
            temperatura = requests.get(f"https://api.openweathermap.org/data/2.5/weather?q={ciudad}&appid={API_KEY_OPENWEATHER}&units=metric").json()["main"]["temp"]
            ciudades[i] = f"('{ciudad}',{round(temperatura)})"
```

5. Encriptación

La encriptacion de la comunicación se ha protegido usando AES.

El Advanced Encryption Standard, abreviado AES, se usa con el fin de cifrar datos y de protegerlos contra cualquier acceso ilícito. El método criptográfico emplea para este objetivo una clave de longitud variada y se denomina según la longitud de clave usada AES-128, AES-192 o AES-256.

Originalmente, este método fue impulsado por el instituto norteamericano National Institute of Standards and Technology y puede emplearse en los EE.UU para cifrar los documentos con clasificación de seguridad máxima.

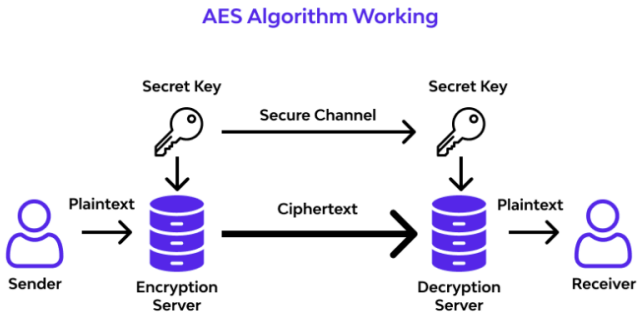
Este método se considera muy seguro y eficiente y sirve para cifrar los datos de todo tipo y por eso, se suele usar en varios protocolos y técnicas de transmisión. P.ej. la protección WPA2 de las redes WiFi utiliza también el Advanced Encryption Standard como p. ej. el estándar SSH o IPsec. Frecuentemente se usan los métodos AES también en la telefonía a través del internet (Voice over IP, VoIP) con el fin de asegurar los datos de señalización o también los datos útiles.

Entretanto, muchos aparatos han integrado permanentemente el Advanced Encryption Standard en su hardware posibilitando cifrados y descifrados mucho más rápidos y eficientes de lo

que se podría hacer si se emplearan meras soluciones de software.

Un motivo para la gran popularidad y propagación de este estándar de cifrado es el libre uso del método sin tarifas de licencia u otras limitaciones asociadas con la patente. Además, los requisitos referentes al hardware y al almacenamiento son relativamente bajos. Posee un algoritmo de cifrado descomplicado y muy elegante cuando se tiene que programar y es fácil de implementar.

El Advanced Encryption Standard usa el denominado algoritmo Rijndael en combinación con el cifrado bloque simétrico como método de cifrado. Las longitudes de bloque y de clave están definidas respectivamente. De este modo, la longitud de bloque es p.ej. de 128 bit y la longitud de clave es de 128, 192 ó 256 bit.



Para implementar AES se han usado las siguientes librerias:

```
#----Práctica 3------ AES
from Crypto.Random import get_random_bytes
from Crypto.Protocol.KDF import PBKDF2
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import Crypto
```

Para la encriptación/desencriptación y creación de la key de AES se han creado los siguientes métodos de la imagen. Al iniciar Engine se crea la clave y se guarda en un archivo '.bin' , y para la desencriptación/encriptación del mensaje se lee ese archivo, por lo que si se requiere cambiar la clave que está en el archivo por otro es posible hacerlo. Esta clave que está en el archivo únicamente se usa para la comunicación en kafka, enviándose ésta mediante socket cuando el player se loguea en la partida. Para la comunicación en sockets y API se ha usado AES también pero dado que la comunicación solo dura 2 mensajes y luego se cierra el canal pues se ha optado por una clave AES provisional ,es decir , se envia en el primero mensaje la clave junto al mensaje cifrado con esa misma clave y al cerrarse la comunicación se borra.

```python
#Crea las claves para usar con el algoritmo de encriptación de mensajes RSA
def crear_clave_AES():
    global aes_kafka_KEY

    #el salt es una combinación de bytes random (necesaria para crear la key)
    salt = b'\x1a\xb3m\xa0\xbf!\xfc\xac6\x7f"\x9f}\xc3\xe0M\xe7\xc4\x94\xc6s\xfc\xa8\x08\xa6C\x19\x9d\x90\xb6\xec\xf3'
    password="vadim" #un password elegido (necesario para crear la key)

    #creo la key
    aes_kafka_KEY = PBKDF2(password,salt,dkLen=32)

    #meto la key dentro de un archivo
    with open("kafka_aes_key.bin", "wb") as f:
        f.write(aes_kafka_KEY)

    print("Clave de encriptación AES para kafka creada. <kafka_aes_key.bin>")
```

```python
def kafka_encriptar(mensaje) -> str:

    with open("kafka_aes_key.bin", "rb") as f:
        kafka_aes_key = f.read()

    msg = bytes(str(mensaje).encode())

    cipher = AES.new(kafka_aes_key,AES.MODE_CBC)
    mensaje_encriptado = cipher.encrypt(pad(msg, AES.block_size))

    return str(list(cipher.iv)) + "|@|" + str(list(mensaje_encriptado))
```

```python
def kafka_desencriptar(mensaje) -> str:

    with open("kafka_aes_key.bin", "rb") as f:
        kafka_aes_key = f.read()

    iv = bytes(eval(mensaje.split("|@|")[0].encode()))
    mensaje_encriptado = bytes(eval(mensaje.split("|@|")[1]))

    cipher = AES.new(kafka_aes_key,AES.MODE_CBC, iv = iv)
    return unpad(cipher.decrypt(mensaje_encriptado), AES.block_size).decode()
```

A parte de AES, en la comunicación API, se ha usado el certificado SSL por lo que se han creado 2 claves más que se guardarán en 1 archivo para cada una. Salta un warning al acceder de que la comunicación no es seguro pero solo es porque no somos una empresa certificadora validada. Se ha utilizado ssl en los archivos 'API Engine' y 'API Registry'

Para la encriptación de contraseñas en la base de datos se ha optado por guardarlas con HASH. Para ellos se ha utilizado bcrypt que realiza esta función.

Aquí un ejemplo de utilización de bcrypt. Antes de hacer el insert hasheo la contraseña, lo que me dará una variable en bytes, la cual convierto en lista y meto en la base de datos, la idea de conviertirlo en una lista es que luego evitar problemas de formato al convertirla en bytes al sacarlo de la base de datos.L uego al leer la contraseña cuando se requiere comprobar si el usuario ha introducido una contraseña correcta se saca esa lista y se convierte en bytes de nuevo y se comprueba con checkpw junto a la contraseña a comprobar en formato bytes.

```python
import bcrypt
```

```python
def insertar_Jugador(self, alias,password,nivel,ef,ec,posicion):
    try:
        hashed_password = bcrypt.hashpw(str.encode(password), bcrypt.gensalt())
        formato_hashed_password = str(list(hashed_password))

        self.cur.execute(f"""insert into jugadores ("alias","password","nivel","ef","ec","posicion") values ('{alias}','{formato_hashed_password}',{nivel},{ef},{ec},'{posicion}')""")
        self.conn.commit() #guardo la info
```

```python
def password_ok(db_hash_lista_password : str , password : str) -> bool:

    if bcrypt.checkpw(str.encode(password), bytes(eval(db_hash_lista_password))): # bytes vs bytes
        return True  #Contraseña correcta
    else:
        return False #Contraseña no correcta
```

6. Despliegue del juego

Para iniciar el despliegue de todos los componentes de la práctica se realizan los siguientes comandos:

```
vadym@vadym-VirtualBox:~/Escritorio/SD P3$ sudo docker-compose up
Creating network "sdp3_default" with the default driver
Creating postgres   ... done
Creating zookeeeper ... done
Creating kafka      ... done
Creating adminer    ... done
```

```
vadym@vadym-VirtualBox:~/Escritorio/SD P3$ python3 db.py
Tabla 'jugadores' creada.
Tabla 'ciudades' creada.
Tabla 'mapa' creada.
Tabla 'jugadores_partida' creada.
Tabla 'ciudades' creada.
```

```
vadym@vadym-VirtualBox:~/Escritorio/SD P3$ python3 API_Engine.py
 * Serving Flask app 'API_Engine'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a product
 * Running on https://192.168.22.100:4000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 317-360-036
```

```
vadym@vadym-VirtualBox:~/Escritorio/SD P3$ python3 API_Registry.py
 * Serving Flask app 'API_Registry'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a producti
 * Running on https://192.168.22.100:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 317-360-036
```

```
vadym@vadym-VirtualBox:~/Escritorio/SD P3/sdfront$ npm run dev -- --host

> sdfront@0.0.0 dev
> vite --host


  VITE v4.0.0  ready in 227 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: http://192.168.22.100:5173/
  ➜  Network: http://10.0.3.15:5173/
  ➜  Network: http://172.18.0.1:5173/
  ➜  press h to show help
```

```
vadym@vadym-VirtualBox:~/Escritorio/SD P3$ python3 AA_Registry.py
[AA_Registry] Servidor a la escucha en ('localhost', 8100)
```

```
vadym@vadym-VirtualBox:~/Escritorio/SD P3$ python3 AA_Engine.py
 [AA_Engine SERVER] Servidor a la escucha en ('localhost', 8202)
Clave de encriptación AES para kafka creada. <kafka_aes_key.bin>
        ######### [MENU] Escoge opción ##########
        1.Nueva partida. 2.Comenzar partida. 3.Parar partida.
```

Una vez están ejecutandose todos los servidores empiezo la partida dándole a '1' en 'AA Engine' lo cual dejará unirse a los jugadores. Luego se abren ventanas de consola con 'AA Player', donde mínimo son 2 jugadores para que empieze la partida. Con el player se une a la partida eligiendo la opción 5 'Unirse a la parrtida' , se introduce alias y password y si todo es correcto se une a la partida y espera a 'AA Engine' comienze la partida. Para comenzar la partida en el 'AA Engine' se escoge la opción 2 del menú y empezará la partida:

```
vadym@vadym-VirtualBox:~/Escritorio/SD P3$ python3 AA_Engine.py

 [AA_Engine SERVER] Servidor a la escucha en ('localhost', 8202)

Clave de encriptación AES para kafka creada. <kafka_aes_key.bin>

        ######### [MENU] Escoge opción ##########

        1.Nueva partida. 2.Comenzar partida. 3.Parar partida.

Opción: 1
Partida iniciada. Esperando jugadores...
```

```
vadym@vadym-VirtualBox:~/Escritorio/SD P3$ python3 AA_Player.py
¡BIENVENIDO AL JUEGO!
Seleccione una opcion:
1. Crear perfil (SOCKETS)
2. Crear perfil (API)
3. Editar un perfil existente (SOCKETS)
4. Editar un perfil existente (API)
5. Unirse a la partida
6. Salir
5
Introduzca su alias:
alicia
Introduzca su password:
pass
Clave AES para kafka recibida.
Identificación correcta. Esperando a que el Engine inicie partida...
```

```
vadym@vadym-VirtualBox:~/Escritorio/SD P3$ python3 AA_Player.py
¡BIENVENIDO AL JUEGO!
Seleccione una opcion:
1. Crear perfil (SOCKETS)
2. Crear perfil (API)
3. Editar un perfil existente (SOCKETS)
4. Editar un perfil existente (API)
5. Unirse a la partida
6. Salir
5
Introduzca su alias:
vadim
Introduzca su password:
pass
Clave AES para kafka recibida.
Identificación correcta. Esperando a que el Engine inicie partida...
```

```
        ######### [MENU] Escoge opción ##########

        1.Nueva partida. 2.Comenzar partida. 3.Parar partida.

Opción: [AA_Engine SERVER] El jugador 'vadim' se ha unido a la partida.
[AA_Engine SERVER] El jugador 'alicia' se ha unido a la partida.
2
```

Al comenzar se le enviará el mapa a los jugadores y se podrá acceder al mapa en el navegador con la ip de la máquina y el puerto del servidor de svelte: