# API REST

1º Creamos un archivo requirements.txt donde irán las librerías que se usarán, se van añadiendo conforme se van instalando. Usamos el comando:

Pip install -r .\requirements.txt

2º Creamos el archivo app.py. Definimos la aplicación y con el comando .run hacemos que solo con el comando python app.py la aplicación corra.

3º Lo siguiente será introducir rutas con app.route y la función que devuelva un json.

from flask import Flask, jsonify

app=Flask(\_\_name\_\_)

@app.route('/hola', methods=['GET'])

def hola():

    return jsonify({"mensaje":"Endpoint desde hola"})

if \_\_name\_\_=="\_\_main\_\_":

    app.run(debug=True)

Lanzamos el servicio y desde postman hacemos la petición GET



Forma

Descripción generada automáticamente con confianza baja

4º Conexión a la base de datos postgre.

Creamos una base de datos con los siguientes comandos:

Create database flask\_db

Para conectarlo tendremos que instalar la librería sqlalchemy y crear un archivo llamado db.py donde introduciremos los parámetros necesarios para establecer la conexión.

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

Base = declarative\_base()

engine = create\_engine("postgresql://postgres:admin@localhost:5432/flask\_db")

Session = sessionmaker(bind=engine)

Lo siguiente será crear los modelos en el archivo models.py

from sqlalchemy import Column, String, Integer

from db import Base, engine

class Usuario(Base):

    \_\_tablename\_\_='usuario'

    id = Column(Integer, autoincrement=True, primary\_key=True)

    username = Column(String(70), unique=True)

    password = Column(String(70))

Base.metadata.create\_all(engine)

Importamos las librerías en el archivo app.py y ejecutamos el proceso

from db import Session, engine

from models import Usuario

5º Creamos un usuario, para ello haremos un endpoint en el archivo app.py

@app.route('/create\_user', methods=['POST'])

def create\_user():

    with engine.connect() as con:

        nuevo\_usuario = Usuario(username="jose", password="123")

        session.add(nuevo\_usuario)

        session.commit()

    return jsonify({"respuesta":"Usuario creado correctamente"})

Si hacemos el post nos dice el usuario creado correctamente, para evitar que nos de error si los parámetros únicos se repiten hacemos un pequeño cambio en el commit()

        try:

            session.commit()

        except:

            return jsonify({"respuesta":"Usuario ya creado en la base de datos"})

Lo siguiente que realizaremos será enviar un json con los parámetros desde el body de postman

Imagen que contiene Aplicación

Descripción generada automáticamente

Si ponemos un print justo debajo de la función:

    print(request)

    print(dir(request))

El dir nos dirá los métodos del request, usaremos el request.data para obtener la información

    data=request.data

    print(data)

    print(type(data))

Observamos que no lo podemos leer, para ello usaremos el método importado json

    data=json.load(request.data)

    print(data)

    print(type(data))

Cuando se realice el post, observamos que recibimos un diccionario, por lo que el código quedará:

@app.route('/create\_user', methods=['POST'])

def create\_user():

    data=json.load(request.data)

    if 'username' not in data:

        return jsonify({"respuesta":"No estás enviando el username"})

    if 'password' not in data:

        return jsonify({"respuesta":"No estás enviando el password"})

    if len(data['username']) == 0:

        return jsonify({"respuesta":"Username no puede estar vacío"})

    if len(data['password']) == 0:

        return jsonify({"respuesta":"password no puede estar vacío"})

    with engine.connect() as con:

        nuevo\_usuario = Usuario(username=data['username'], password=data['password'])

        session.add(nuevo\_usuario)

        try:

            session.commit()

        except:

            return jsonify({"respuesta":"Usuario ya creado en la base de datos"})

    return jsonify({"respuesta":"Usuario creado correctamente"})

6º Lo siguiente será crear un nuevo modelo, para ello lo primero que debemos hacer es importar Foreign\_key de sqlalchemy

from sqlalchemy import Column, String, Integer

from db import Base, engine

from sqlalchemy.schema import ForeignKey

from sqlalchemy.orm import relationship

class Usuario(Base):

    \_\_tablename\_\_='usuario'

    id = Column(Integer, autoincrement=True, primary\_key=True)

    username = Column(String(70), unique=False)

    password = Column(String(70))

    ventas = relationship('Ventas', backref="usuario")

class Ventas(Base):

    \_\_tablename\_\_='ventas'

    id = Column(Integer, autoincrement=True, primary\_key=True)

    username\_id=Column(Integer, ForeignKey("usuario.id"))

    venta=Column(Integer)

Base.metadata.create\_all(engine)

Creamos una nueva ruta en el app.py, se tratará de un método GET y servirá para obtener la información de los usuarios que tienen ventas asignadas. Para ello, se le pasa un objeto.

@app.route('/obtener\_venta', methods=['GET'])

def obtener\_venta():

    data=json.loads(request.data)

    if 'username' not in data:

        return jsonify({"respuesta": "Username no enviado, validar datos!"})

    with engine.connect() as con:

        obtener\_usuario = f"select \* from usuario where username = '{data['username']}'"

        respuesta = con.execute(obtener\_usuario).one()

        obtener\_venta=f"select venta from ventas where username\_id = '{respuesta[0]}'"

        respuestas\_ventas = con.execute(obtener\_venta)

        respuestas\_ventas=[i[0]for i in respuestas\_ventas]

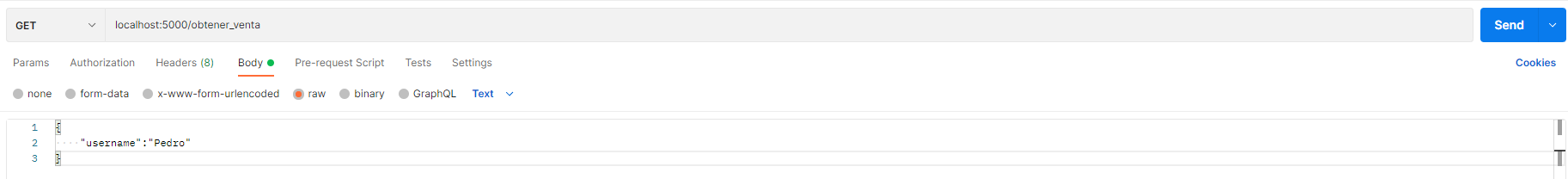
        return jsonify({"ventas\_usuario":{"usuario":data['username'], "ventas":respuestas\_ventas}})

Para probar, se le introduce manualmente las ventas en SQL

Texto

Descripción generada automáticamente con confianza baja

Con la siguiente consulta se obtienen datos:



7º Se le añadirá hash a la contraseña, para ello se tendrá que modificar los modelos y aumentar el tamaño de los string. Se eliminará la base de datos, para poder subirla. Otra forma sería con migraciones pero se verá más adelante.

Se modifica esta parte del modelo

password = Column(String(200))

Se vuelve a ejecutar el comando y se obtiene la nueva base de datos.

Se importará un módulo llamado wekzeug a través del cual encriptaremos la contraseña

from werkzeug.security import generate\_password\_hash

Y donde antes se pasaba la contraseña ahora pasaremos lo siguiente en la función crear\_usuario.

        hash\_password=generate\_password\_hash(data['password'], method="sha256")

        nuevo\_usuario = Usuario(username=data['username'], password=hash\_password)

7º Protección de la API con la generación de JWT. Para ello, se creará un método que reciba un usuario y una contraseña. Si ambos son correctos, se generará un token.

Importamos las siguientes librerías (incluido jwt, la cual se tiene que importar en el requirements):

from werkzeug.security import generate\_password\_hash, check\_password\_hash

from functools import wraps

import jwt

import datetime

Lo siguiente sería el decorador, el token\_required, esta función lo que realizará será añadir el decorador a las funciones

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Lo siguiente será añadir el decorador antes de coger los datos. Este decorador lo que hace es inicializar una variable llamada token a None, pregunta si x-head está en los headers que se envían desde postman (tenemos que crear un parámentro). Después intenta validarlo para ver si el token está activo.

Para crear usuarios, tendremos que gestionar unos permisos para que solo puedan algunos. Para ello, cuando codificamos en data, vemos que es un diccionario y que tiene el identificador public\_id. Podemos acceder a él. Actualmente cualquier persona puede crear usuarios.

Para evitar eso, creamos uno llamado administrador.

Para detectar que public\_id es el que está entrando en la aplicación, mandamos este parámetro y tendrá que ser recogido por la función donde se use el decorador

def obtener\_venta(current\_user):

En cualquier función donde se use el decorador debe recibir el usuario.

Añadiremos el decorador a la función create\_user() para limitar los usuarios que tengan permisos para crear.

8º Migraciones SQLAlchemy.

Si nosotros queremos agregar columnas a una base de datos, no se modificarán sino utilizamos flask\_migrate.

Se puede comentar la última línea de los modelos debido a que no se utilizará.

En el app.py se tendrá que introducir cambios

Lo primero que se tendrá que crear es el archivo manage.py y copiamos el siguiente archivo. Antes que nada, hay que añadir flask\_required y flask\_migrate y flask\_sqlalchemy.

En el archivo db.py creamos una variable con la conexión y la pasamos en el engine y también al app.config

Tenemos que modificar los modelos para tenerlo en cascada

Los modelos se tienen que importar después de inicializar la base de datos en app.py

Ahora si queremos añadir una nueva columna

9º Lo siguiente que se realizará será crear la base de datos en heroku y conectarla a DBeaver. Primero tenemos que registrarnos, después pinchamos en create new app, introducimos nombre: proyecto-flask-api-rest. En resource crearemos nuestra base de datos, buscamos postgre y pinchamos en heroku postgre, pinchamos en el plan gratis. Para conectar la base de datos al DBeaver, primero debemos instalarlo.

Vamos de nuevo a heroku y al recurso creado, vamos a settings y despues a database credential donde estarán los detalles

En dbeaver, pinchamos en nueva conexión y postgre. En el host copiamos el que nos indica las credenciales, después el nombre de la base de datos, el puerto es el indicado, en nombre de usuario copiamos el user, la contraseña en la password y probamos la conexión.

Lo siguiente será cambiar las conexiones de nuestra aplicación, en concreto, en el archivo db.py modificamos la extensión

connection\_db = "postgresql://dzzjehwhljgrqd:c2daab1612c9f5b82a2f432c91969d01e0aade35ac69cae9f2371fe2aa105188@ec2-52-49-120-150.eu-west-1.compute.amazonaws.com:5432/d1o747k2054ltn"

Para desplegar crearemos el archivo Procfile y añadimos:

web: gunicorn app:app

tendremos que añadir a los requirements la librería gunicorn.

Lo siguiente será instalar el CLI de heroku, abrimos el cmd y ponemos heroku. El siguiente comando heroku login (presionamos enter) y tendremos que logearnos para verificar que somos nosotros

9º Creación swagger – blueprint

En un swagger, tendremos las distintas direcciones de la API, en ellos se incluirán los campos requeridos y los que no.

Lo primero que se hará será crear una carpeta llamada API y app.py solamente renderizará templates. En la carpeta API, se crearán dos carpetas, una controlores.py y otra functions.py.

Comenzaremos con los controladores, se utilizará blueprint, jsonify y request. Para la validación de los campos, se utilizará flask\_restplus (se debe añadir al requirement) para importar Api, resource y fields.

Rellenamos los parámetros necesarios para la API en el controlador. Lo siguiente será introducir bp\_api en app.py. Para ello, introducimos lo siguiente:

from api.controllers import bp\_api

app.register\_blueprint(bp\_api)

Si nos vamos a la dirección <http://localhost:5000/Api/> veremos que está creado el swagger con las características indicadas en el controller.py

Lo siguiente será pasar el login, para ello crearemos una clase en el controllers.py

Para simplificar el código de login introducido en app.py crearemos una función para verificar el usuario.

Para incluir que username y password son campos requeridos, se creará otra clase que servirá para validar los datos:

class VerificacionDatos():

    Login = api.model('login',{

        "username":fields.String(description=u"username", required=True),

        "password":fields.String(description=u"password", required=True)

    })

Se tendrá que incluir en la clase que se quiere verificar justo antes de la función con la máscara:

@api.expect(VerificacionDatos.Login, Validate=True)

También hay que añadir una ruta a la clase y una documentación a la ruta, quedando:

@api.route('/Login/')

@api.doc(description="Correo y contraseña")

class Login(Resource):

    @api.expect(VerificacionDatos.Login, Validate=True)

    def post(self):

        auth=request.json

        valida = valida\_user(auth["username"],auth["password"])

        if 'token' in valida:

            return jsonify(valida)

        return jsonify({"Respuesta":"Login requerido"})

Lo siguiente que se hará, será coger los datos como un request. Otra cosa que se realizará será controlar el .one() para cuando viene vacía la consulta. El archivo de function.py quedaría:

from werkzeug.security import generate\_password\_hash, check\_password\_hash

import datetime

from functools import wraps

from flask\_sqlalchemy import SQLAlchemy

import jwt

from db import Session, engine

llave='Th1s1ss3cr3t'

def valida\_user(username, password):

    with engine.connect() as con:

        try:

            user = con.execute(f"select \* from usuario where username = '{username}' ").one()

        except:

            user = None

    if user:

        if check\_password\_hash(user[2], password):

            token = jwt.encode({'public\_id':user[1], 'exp': datetime.datetime.utcnow()+datetime.timedelta(minutes=30)}, llave)

            return {'token':token.decode('UTF-8')}

    return {"respuesta":"Contraseña incorrecta"}

Lo siguiente será declarar el ns\_model de la siguiente manera, ya no será el decorador de api sino el de ns\_model. Con esto cambiamos el nombre de la cabecera

from urllib import request

from flask import Blueprint, jsonify, request

from flask\_restplus import Api, Resource, fields

from api.functions import \*

bp\_api = Blueprint('Api',\_\_name\_\_,url\_prefix="/Api")

api = Api(bp\_api, version="1.0", title="Api",description="End Points")

ns\_model = api.namespace('Methods', description='Metodos')

class VerificacionDatos():

    Login = api.model('login',{

        "username":fields.String(description=u"username", required=True),

        "password":fields.String(description=u"password", required=True)

    })

@ns\_model.route('/Login/')

@api.doc(description="Correo y contraseña")

class Login(Resource):

    @ns\_model.expect(VerificacionDatos.Login, Validate=True)

    def post(self):

        auth=request.json

        valida = valida\_user(auth["username"],auth["password"])

        if 'token' in valida:

            return jsonify(valida)

        return jsonify({"Respuesta":"Login requerido"})

Con las validaciones, conseguimos eliminar todos los if al obligarlos a ser campos requeridos.

# APLICACIÓN

1º En este punto veremos la parte visual. Necesitaremos crear una carpeta de templates para meter los archivos html. En el archivo app se importará render\_template.

El primer archivo template que se creará será el base.html, servirá de base para los siguientes y el código es:

<!DOCTYPE html>

<html lang="en">

    <head>

        <meta charset="UTF-8">

        <meta http-equiv="X-UA-Compatible" content="IE-edge">

        <meta name="viewport" content="width-device-width, initial-scale=1.0">

        <title>{% block title %}{% endblock %}</title>

    </head>

    <body>

        {% block body %}{% endblock %}

    </body>

</html>

También se creará un index.html:

{% extends 'base.html' %}

{% block title %}

Index

{% endblock %}

{% block body %}

    <h1>HOLA DESDE EL INDEX</h1>

{% endblock %}

Para renderizar estas plantillas, en la ruta raíz se incluye el renderizado

@app.route('/', methods=['GET'])

def index():

    return render\_template('index.html')

Para utilizar plantillas de Bootstrap, nos metemos dentro de la documentación. Tenemos que añadirle el link en el archivo base para que nos reconozca las clases.

{% extends 'base.html' %}

{% block title %}

Registro Usuario

{% endblock %}

{% block body%}

    <form method="POST" action="/registro">

        <label for="">Registro Usuario</label>

        <div class="form-group">

            <span>Usuario</span>

            <input type="text" class="form-control" id="username" placeholder="Usuario" name="username" required>

        </div>

        <div class="form-group">

            <span>Contraseña</span>

            <input type="password" class="form-control" id="password" placeholder="Contraseña" name="password" required>

        </div>

        <button type="submit" class="btn btn-primary"> Registrar</button>

    </form>

{% endblock %}

@app.route('/registro', methods=['GET','POST'])

def registro():

    if request.method=='POST':

        username=request.form['username']

        password=request.form['password']

        print(f'username es {username} y password es {password}')

        return render\_template('index.html')

    return render\_template('registro.html')

Lo siguiente será consumir del endpoint, concretamente se consumirá de la api que genera el token y guardará JWT en sesión.

La intención será que un usuario que esté registrado pueda entrar al index.html. Tendremos que generar un nuevo archivo llamado login.html que será muy similar a la de registro.html

{% extends 'base.html' %}

{% block title %}

Iniciar sesión

{% endblock %}

{% block body%}

    <form method="POST" action="/login">

        <label for="">Iniciar sesión</label>

        <div class="form-group">

            <span>Usuario</span>

            <input type="text" class="form-control" id="username" placeholder="Usuario" name="username" required>

        </div>

        <div class="form-group">

            <span>Contraseña</span>

            <input type="password" class="form-control" id="password" placeholder="Contraseña" name="password" required>

        </div>

        <button type="submit" class="btn btn-primary"> Registrar</button>

    </form>

{% endblock %}

Como se muestra, con las etiquetas span, se recordará que no hay usuario creado

Lo siguiente que se mostrará será la función registro, la cual realiza un post al método create user presente también en el archivo app.py (en el futuro se migrará). Recordar que todas las APIs están alojadas en el controller.py y todas las plantillas renderizadas en la app.py.

@app.route('/registro', methods=['GET','POST'])

def registro():

    if request.method=='POST':

        username=request.form['username']

        password=request.form['password']

        args = {

            "username":username,

            "password":password

        }

        response = requests.post('http://localhost:5000/create\_user', json=args)

        if response.status\_code == 200:

            print('Entre', response.status\_code)

            return render\_template('index.html')

        else:

            return render\_template('registro.html')

    return render\_template('registro.html')

Lo siguiente será registrar el método login en app.py. En este método, se tendrá que hacer un post al validador. Será parecido al anterior pero en este caso, tendremos que consumir de la api creada. Para ello, el post se realizará a la url :

http://localhost:5000/Api

y se le añadirá la ruta que hemos puesto en la cabecera del swagger, para el loggin es:

/Methods/Login/

Una vez hecho esto, tenemos que recuperar el token que nos devuelve la respuesta.

@app.route('/login', methods=['GET','POST'])

def login():

    if request.method=='POST':

        username=request.form['username']

        password=request.form['password']

        args = {

            "username":username,

            "password":password

        }

        response = requests.post('http://localhost:5000/Api/Methods/Login/', json=args)

        if response.status\_code == 200:

            response\_api = json.loads(response.text)

            return render\_template('index.html')

        else:

            return render\_template('login.html')

    return render\_template('login.html')

Lo siguiente será guardar el token en la sesión, modificará token\_required

def token\_required(f):

   @wraps(f)

   def decorator(\*args, \*\*kwargs):

        try:

            data = jwt.decode(session['token'], app.config['SECRET\_KEY'])

        except:

            return redirect(url\_for('login'))

        return f(data['public\_id'],\*args, \*\*kwargs)

   return decorator

Una vez esto hecho habrá que incluir el decorardor en las urls que se quiera validar por el token

Por último, se realizará el logout:

@app.route('/logout', methods=['GET' ])

def logout():

    session.clear()

    return render\_template('login.html')

Lo siguiente será consumir la api de crear usuario que actualmente estaba en app.py. Lo siguiente será modificar todas las rutas para incluirlas en la API.

Las querys que se realizarán se incluirán el un nuevo archivo llamado query.py

Lo último que se verá en esta sección serán los métodos patch y delete

Una vez creado varias URLs para el swagger, lo siguiente que se realzará un formulario para crear ventas.

Si al hacer un login, el usuario no existe no nos puede devolver un 200, eso se solucionará utilizando un abort y usándolo en los controllers.

Para el tema de los roles se añadirá una nueva columna desde los modelos, en los usuarios creamos la columna rol.

Tenemos que realizar las migraciones. Posteriormente, se le asignará el rol admin a alguno.

Primero tendremos que encargar un endpoint que se encargue de validar el rol y se encargue de almacenarlo en la sesión al igual que el token.

# REDIS

Redis servirá como una base de datos, en concreto se utilizará para evitar que se inicie sesión con otro navegador provocando problemas.

Lo primero que se realizará será la instalación. Entramos al repositorio de Git, [https://github.com/microsoftarchive/redis/releases después descargamos el .zip](https://github.com/microsoftarchive/redis/releases%20después%20descargamos%20el%20.zip) [**Redis-x64-3.0.504.zip**](https://github.com/microsoftarchive/redis/releases/download/win-3.0.504/Redis-x64-3.0.504.zip), posteriormente lo descomprimimos, en nuestro caso lo almacenamos en la carpeta.

Lo siguiente será configurar las variables de entorno. Buscamos variables de entorno, después hacemos click en variables de entorno, click en editar (con PATH pinchado), aquí colocamos la ruta donde se descomprimió el zip, en nuestro caso: C:\Users\josep\Desktop\proyecto\_rest\_API\_Flask\_Completo\_Redis\Redis-x64-3.0.504 .Pinchamos en nuevo, lo pegamos y aceptar todas las veces.

Podemos introducir algunos comandos:

* Iniciamos servidor: redis-server
* Nos conectamos desde otra consola: redis-cli
* Para saber si estamos bien conectados mandamos: PING y nos devolvería un PONG
* Seteamos a la llave nombre el valor Andres: Set nombre Andres
* Para coger el valor: get nombre

Lo siguiente será integrar redis en una aplicación de flask. Lo que se realizará será cuando un usuario inicia sesión, que mediante otro navegador no se permita abrir otra sesión. Para correr redis ejecutamos:

* redis-server

En el caso que se quiera ver lo guardado ejecutamos el comando

* redis-cli

A partir de este creamos y vemos variables guardadas, como vimos anteriormente.

Lo primero será importar redis, para ello habrá que introducirlo en el requirements.txt e instalamos. Lo importamos en app.py y lo inicializamos.

Crearemos una ruta para ver como funciona. Se creará una llave con un valor y se alojará en redis

Nos devuelve un tipo byte y hay que convertirlo a str.

Lo primero que se tendrá que hacer será guardar en la sesión el username y registrarlo en redis.