Exámen Final - IS

Integrantes:

- Anderson David Cárcamo Vargas
- Mishelle Stephany Villarreal Falcón

Pregunta 1

Código fuente

- Colaboramos en un repositorio en github, para trabajar de manera simultánea

```
https://github.com/AndersonCarcamo/examen2Software/tree/main
```

model.sql

- Hemos creado una base de datos en **postgres** con las tablas necesarias
- Luego insertamos los datos expuestos en el documento

```
CREATE DATABASE billetera_yape;

\c billetera_yape;

CREATE TABLE cuentausuario(
    numero VARCHAR(9) PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    saldo INT NOT NULL
);

CREATE TABLE contacto(
    id SERIAL PRIMARY KEY,
    numero_usuario VARCHAR(9) REFERENCES cuentausuario(numero),
    numero_contacto VARCHAR(9) REFERENCES cuentausuario(numero),
    UNIQUE(numero_usuario, numero_contacto)
);

CREATE TABLE operacion(
    id SERIAL PRIMARY KEY,
    cuenta_origen VARCHAR(9) REFERENCES cuentausuario(numero),
    cuenta_destino VARCHAR(9) REFERENCES cuentausuario(numero),
    valor INT NOT NULL,
    fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

INSERT INTO cuentausuario (numero, nombre, saldo) VALUES
('21345', 'Arnaldo', 200),
('123', 'Luisa', 400),
('456', 'Andrea', 300);

INSERT INTO contacto (numero_usuario, numero_contacto) VALUES
('21345', '123'),
('21345', '456'),
```

```
('123', '456'),
('456', '21345');
```

app.py

- Con app.py podemos conectar con la BD que hemos creado, dentro de la BD se pondrán registrar los datos requeridos

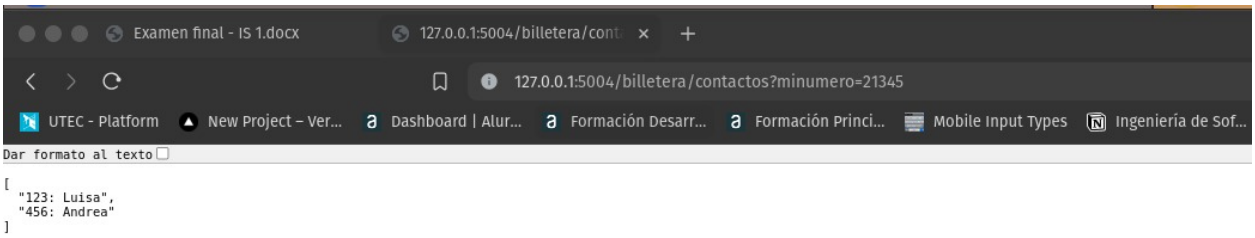
```
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy
from datetime import datetime
from flask_cors import CORS

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://postgres:A
db = SQLAlchemy(app)

CORS(app, resources={r"/*": {"origins": "*"}})

class Cuentausuario(db.Model):
    __tablename__ = 'cuentausuario'
    numero = db.Column(db.String(9), primary_key=True)
    nombre = db.Column(db.String(100), nullable=False)
    saldo = db.Column(db.Integer, nullable=False)

class Contacto(db.Model):
    __tablename__ = 'contacto'
    id = db.Column(db.Integer, primary_key=True)
    numero_usuario = db.Column(db.String(9), db.ForeignKey('cuer
    numero_contacto = db.Column(db.String(9), db.ForeignKey('cue
    __table_args__ = (db.UniqueConstraint('numero_usuario', 'nur

class Operacion(db.Model):
    __tablename__ = 'operacion'
    id = db.Column(db.Integer, primary_key=True)
    cuenta_origen = db.Column(db.String(9), db.ForeignKey('cuent
```

```
    cuenta_destino = db.Column(db.String(9), db.ForeignKey('cuer
    valor = db.Column(db.Integer, nullable=False)
    fecha = db.Column(db.DateTime, default=datetime.utcnow)

@app.route('/billetera/contactos', methods=['GET'])
def listar_contactos():
    minumero = request.args.get('minumero')
    numero = Cuentausuario.query.filter_by(numero=minumero).firs
    if not numero:
        return jsonify({'error': 'Numero no encontrado'}), 404
    contactos = Contacto.query.filter_by(numero_usuario=minumero
    contactos_info = []
    for contacto in contactos:
        contacto_usuario = Cuentausuario.query.filter_by(numero=
        if contacto_usuario:
            contactos_info.append(f"{contacto.numero_contacto}:
    return jsonify(contactos_info), 200

@app.route('/billetera/pagar', methods=['GET'])
def pagar():
    minumero = request.args.get('minumero')
    numero_destino = request.args.get('numerodestino')
    valor = int(request.args.get('valor'))
    cuenta_origen = Cuentausuario.query.filter_by(numero = minum
    if not cuenta_origen:
        return jsonify({'error': 'Este numero no existe'}), 404
    cuenta_destino = Cuentausuario.query.filter_by(numero=numero
    if not cuenta_destino:
        return jsonify({'error': 'Numero destino no existe'}), 4
    if cuenta_origen.saldo < valor:
        return jsonify({'error': 'Saldo insuficiente'}), 400
    # descuenta el saldo en la cuenta de origen
    cuenta_origen.saldo -= valor
    # aumenta el saldo en la cuenta de destino
    cuenta_destino.saldo += valor
    operacion = Operacion(cuenta_origen=minumero, cuenta_destino
```

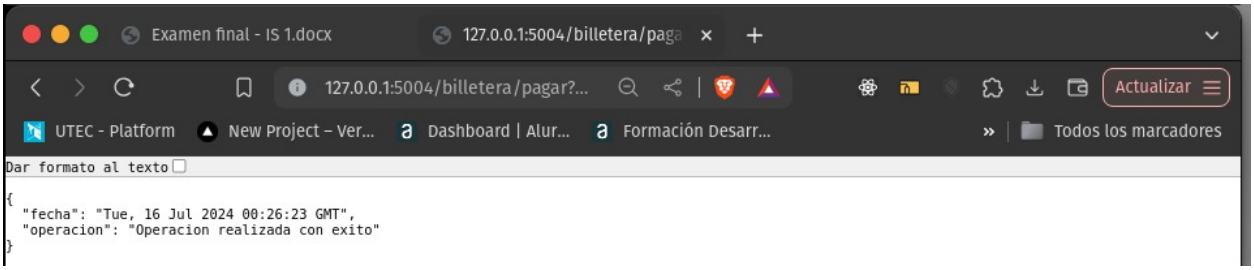
```
    db.session.add(operacion)
    db.session.commit()
    return jsonify({'operacion': 'Operacion realizada con exito

@app.route('/billetera/historial', methods=['GET'])
def historial_operaciones():
    minumero = request.args.get('minumero')
    cuentausuario = Cuentausuario.query.filter_by(numero=minume
    if not cuentausuario:
        return jsonify({'error': 'Este numero no existe'}), 404
    saldo_actual = cuentausuario.saldo
    operaciones_realizadas = Operacion.query.filter_by(cuenta_o
    operaciones_recibidas = Operacion.query.filter_by(cuenta_des
    operaciones_info = []
    for operacion in operaciones_realizadas:
        cuenta_destino = Cuentausuario.query.filter_by(numero=op
        operaciones_info.append(f"Pago realizado de {operacion.v
    for operacion in operaciones_recibidas:
        cuenta_origen = Cuentausuario.query.filter_by(numero=ope
        operaciones_info.append(f"Pago recibido de {operacion.va
    return jsonify({'saldo': saldo_actual, 'operaciones': operac


if __name__ == '__main__':
    with app.app_context():
        db.create_all()  # Crea las tablas en la base de datos s
    app.run(debug=True, host='0.0.0.0', port=5004)
```

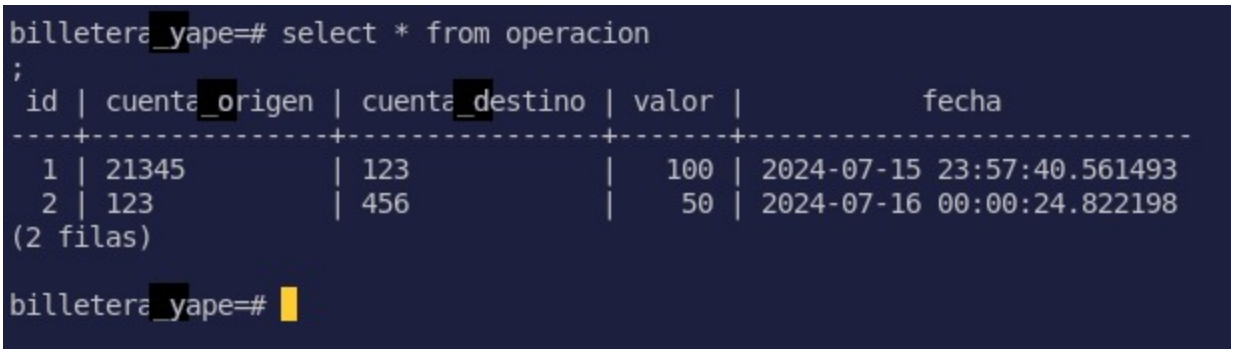
prueba de los endpoints:

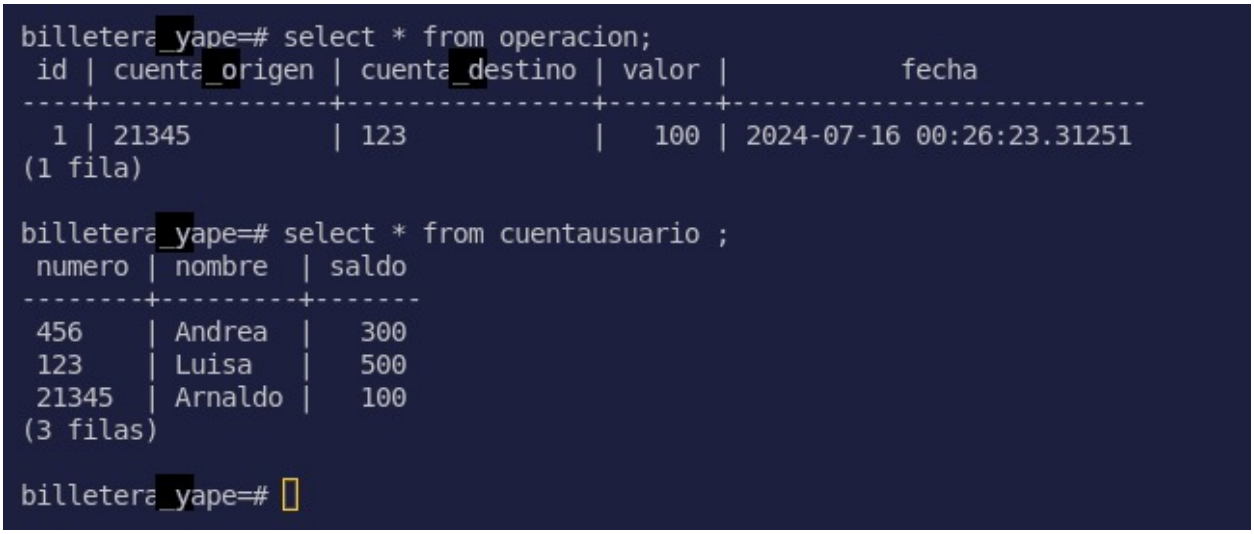
/billetera/contactos?minumero=21345



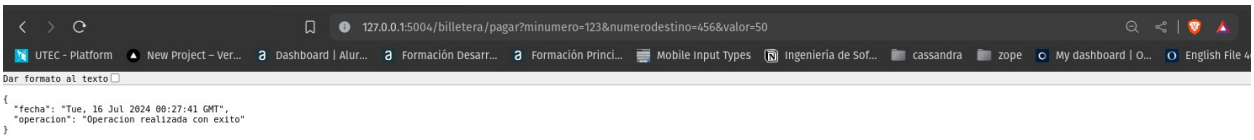
/billetera/pagar?minumero=21345&numerodestino=123&valor=100



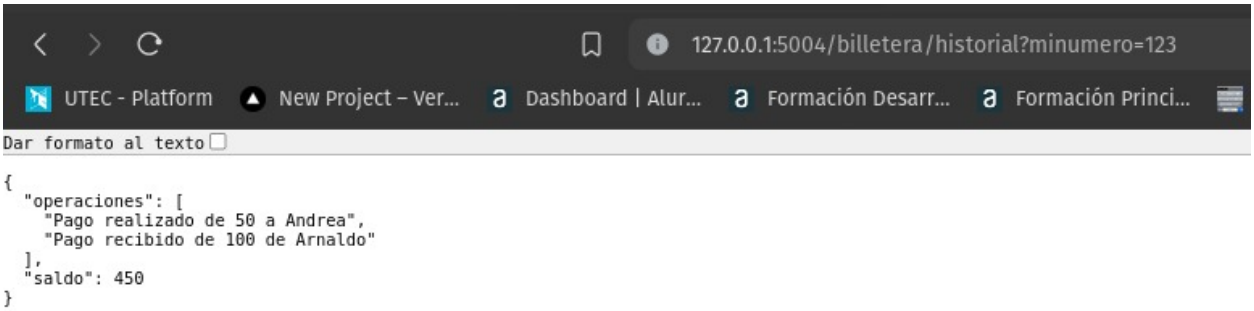




/billetera/pagar?minumero=123&numerodestino=456&valor=50




```
billetera_yape=# select * from operacion;
 id | cuenta_origen | cuenta_destino | valor |          fecha
----+---------------+----------------+-------+----------------------------
  1 | 21345         | 123            |   100 | 2024-07-16 00:26:23.31251
  2 | 123           | 456            |    50 | 2024-07-16 00:27:41.32877
(2 filas)

billetera_yape=# select * from cuentausuario ;
 numero | nombre  | saldo
--------+---------+-------
 21345  | Arnaldo |   100
 123    | Luisa   |   450
 456    | Andrea  |   350
(3 filas)

billetera_yape=#
```



Pregunta 2 - Pruebas unitarias

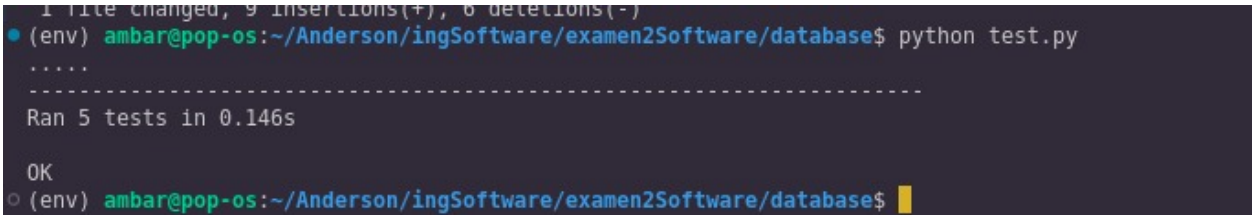
test.py

```
import unittest
from app import app, db, Cuentausuario, Contacto, Operacion
from datetime import datetime

class TestModels(unittest.TestCase):

    def setUp(self):
        # Configurar la aplicación Flask para pruebas
        self.app = app
        self.app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresq
        self.app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = Fals
        self.app.config['TESTING'] = True

        with self.app.app_context():
            db.drop_all()
            db.create_all()
            # Añadir datos iniciales
            self.cuenta1 = Cuentausuario(numero='21345', nombre=
            self.cuenta2 = Cuentausuario(numero='123', nombre='I
            self.cuenta3 = Cuentausuario(numero='456', nombre='A
            db.session.add_all([self.cuenta1, self.cuenta2, self
            db.session.commit()

            self.contacto1 = Contacto(numero_usuario='21345', nu
            self.contacto2 = Contacto(numero_usuario='21345', nu
            db.session.add_all([self.contacto1, self.contacto2])
            db.session.commit()

    def tearDown(self):
        with self.app.app_context():
            db.session.remove()
            db.drop_all()

    # Caso de éxito: Transferencia exitosa entre cuentas
    def test_transferencia_exitosa(self):
```

```
        with self.app.app_context():
            cuenta1 = Cuentausuario.query.filter_by(numero='2134
            cuenta2 = Cuentausuario.query.filter_by(numero='123
            operacion = Operacion(cuenta_origen='21345', cuenta_
            cuenta1.saldo -= 100
            cuenta2.saldo += 100
            db.session.add(operacion)
            db.session.commit()
            self.assertEqual(cuenta1.saldo, 100)
            self.assertEqual(cuenta2.saldo, 600)

    # Caso de éxito: Historial de operaciones
    def test_historial_operaciones(self):
        with self.app.app_context():
            operacion1 = Operacion(cuenta_origen='21345', cuenta
            operacion2 = Operacion(cuenta_origen='21345', cuenta
            db.session.add_all([operacion1, operacion2])
            db.session.commit()
            historial = Operacion.query.filter((Operacion.cuenta
            self.assertEqual(len(historial), 2)
            self.assertEqual(historial[0].valor, 100)
            self.assertEqual(historial[1].valor, 50)

    # Caso de error: Transferencia a un contacto que no está en
    def test_transferencia_error_contacto_no_en_lista(self):
        with self.app.app_context():
            with self.assertRaises(ValueError) as context:
                operacion = Operacion(cuenta_origen='21345', cue
                if '100' not in [contacto.numero_contacto for co
                    raise ValueError("El contacto no está en la
            self.assertTrue('El contacto no está en la lista de

    # Caso de error: Transferencia con saldo insuficiente
    def test_transferencia_error_saldo_insuficiente(self):
        with self.app.app_context():
            with self.assertRaises(ValueError) as context:
```

```
                operacion = Operacion(cuenta_origen='21345', cue
                cuenta1 = Cuentausuario.query.filter_by(numero=
                cuenta1.saldo = 200
                if cuenta1.saldo < 300:
                    raise ValueError("Saldo insuficiente.")
            self.assertTrue('Saldo insuficiente.' in str(contex

    # Caso de error: Transferencia a una cuenta inexistente
    def test_transferencia_error_cuenta_inexistente(self):
        with self.app.app_context():
            with self.assertRaises(ValueError) as context:
                cuenta_destino = Cuentausuario.query.filter_by(
                if not cuenta_destino:
                    raise ValueError("Cuenta destino no encontra
            self.assertTrue('Cuenta destino no encontrada.' in s

if __name__ == '__main__':
    unittest.main()
```

Resultado:



Pregunta 3

Se requiere realizar un cambio en el software para que soporte un valor máximo de 200 soles a
transferir por día

> Qué cambiaría en el código (Clases / Métodos) - No realizar la implementación, sólo

descripción.

- Primero que nada hay que añadir un campo en la tabla ‘cuentausuario’ llamado limit que sea un integer que almacene cuanta cantidad de saldo ha sido usado.
- Cada fin de día todos los valores limit se deben reiniciar a 0. Esto lo podemos manejar desde la base de datos con una function and trigger y, desde el código de endpoint con una función que itere todos las cuentas y resetee el limit.
- En la función de pago realizar una verificación de que el pago no exceda un valor de 200. Caso contrario cortar la función ahí.
- Cada vez que se realiza un pago en la función de pago. Hay que aumentar el limit de la cuentausuario que realiza el pago.
- Verificar que dicho limit no haya excedido los 200 soles. Si es que ha excedido votar un json de {’error’: ‘limite diario excedido’}. Caso contrario continuar con el flujo de un pago.

Qué casos de prueba nuevos serían necesarios?

- Verificar que el limit para todos las cuentas de usuario se han reiniciado a 0 a inicio del día.
- Verificar que el limit está aumentando cada que se realiza un pago
- Verificar que los saldos de las cuentas de origen y destino en el pago no han sido modificadas cuando el limit ha sido excedido.

Los casos de prueba existentes garantizan que no se introduzcan errores en la funcionalidad existente?

- Si, porque tenemos test para transferencia entre cuentas y verifica que sean dentro de los contactos también con saldo suficientes y que sea una cuenta existente