SPRINT DELIVERY- 4

Date	17th November 2022
Team ID	PNT2022TMID28587
Project name	Smart Waste Management System for Metropolitan Cities

APP→API→ROUTES.PY:

```
from flask import Blueprint, jsonify, request, abort, Response, send_file

from app.models import Basket, User, Waste, Vehicle, Employee, commit, Area, SoftwareVersion,
BasketType

from app.validate import validate

import json

from datetime import datetime

from io import BytesIO


api = Blueprint('api', __name__, url_prefix='/api')



@api.route('/')

def index():

    return jsonify({"message": "the api is working"})



@api.route('baskets')

def get_baskets():

    baskets = Basket.query.all()

    baskets_list = [basket.format() for basket in baskets]

    return jsonify({

        "baskets": baskets_list,

        "total_baskets": len(baskets_list)

    })
```

```
@api.route('baskets/<int:basket_id>')
def get_basket(basket_id):
    basket = Basket.query.get(basket_id)
    return jsonify({
        "basket": basket.format()
    })


@api.route('baskets/<int:basket_id>/wastes')
def get_wastes_of_basket(basket_id):
    basket = Basket.query.get(basket_id)
    wastes = [waste.format() for waste in basket.wastes]
    total_size = 0.0
    for waste in wastes:
        total_size = total_size + waste['size']
        print(type(waste['size']), waste['size'])
        print(total_size)
    print(total_size)
    return jsonify({
        "basket_id": basket.id,
        "total_size": total_size,
        "wastes": wastes
    })


@api.route('baskets', methods=['POST'])
def add_new_basket():
    data = request.json
    roles = ['longitude', 'latitude', 'area_code']
    abort(400) if not validate(roles, data) else None
    longitude = data['longitude']
    latitude = data['latitude']
    area_code = data['area_code']
    type_id = data['type']
    area = Area.query.get(area_code)
    if not area:
        abort(422)
    basket_type = BasketType.query.get(type_id)
    if not basket_type:
        abort(422)
    basket = Basket(longitude=longitude, latitude=latitude, area=area, basketType=basket_type).save()

    return jsonify({
        "success": True,
        "basket": basket.format()
    })


@api.route('baskets/<int:basket_id>', methods=['DELETE'])
def delete_baskets(basket_id):
    basket = Basket.query.get(basket_id)
    basket = basket.delete() if basket else basket
    return jsonify({
        'success': bool(basket)
    })


@api.route('baskets', methods=['PATCH'])
def update_all_baskets():
    data = request.json
    software_version = data['software_version']
    baskets = Basket.query.update({Basket.software_version: software_version})
    commit()
    return jsonify({
        "baskets_update": baskets
    })



@api.route('baskets/<int:basket_id>', methods=['PATCH'])
def update_the_basket(basket_id):
    data = request.json
    basket_level = data['level']
    if basket_level is None:
        abort(400)
    try:
        basket = Basket.query.get(basket_id)
        basket.wastes_height = basket_level
        basket.save()
    except:
        abort(422)

    return jsonify({
        "success": True,
    })



@api.route('areas')
def get_areas():
    areas = Area.query.all()
    areas_list = [area.format() for area in areas]

    return jsonify({
        "total_areas": len(areas_list),
        "areas": areas_list
    })
```

```
@api.route('areas/<int:area_code>')
def get_area(area_code):
    area = Area.query.get(area_code)
    return jsonify({
        "area": area.format()
    })


@api.route('areas/<int:area_code>/baskets')
def get_basket_belong_to_area(area_code):
    baskets = Area.query.get(area_code).baskets
    baskets_list = [basket.format() for basket in baskets]
    return jsonify({
        "total_baskets": len(baskets_list),
        "baskets": baskets_list
    })


@api.route('areas/<int:area_code>/users')
def get_user_belong_to_area(area_code):
    users = Area.query.get(area_code).users
    users_list = [user.format() for user in users]
    return jsonify({
        "total_users": len(users_list),
        "users": users_list
    })


@api.route('areas', methods=['POST'])
def insert_new_area():
    data = request.json
    roles = ['area_code', 'area_name', 'area_size', 'longitude', 'latitude', 'city']
    if not validate(roles, data):
        abort(422)
    code = data['area_code']
    name = data['area_name']
    size = data['area_size']
    longitude = data['longitude']
    latitude = data['latitude']
    city = data['city']
    area = Area(code=code, name=name, size=size, longitude=longitude, latitude=latitude, city=city)
    area.save(True)
    return jsonify({
        "success": True,
        "area": area.format()
    })


@api.route('vehicles')
def get_vehicles():
    vehicles = Vehicle.query.all()
    vehicles_list = [vehicle.format() for vehicle in vehicles]
    return jsonify({
        "vehicles": vehicles_list
    })


@api.route('vehicles/<int:vehicle_plate_no>')
def get_vehicle(vehicle_plate_no):
    vehicle = Vehicle.query.get(vehicle_plate_no)
    return jsonify({
        "vehicle": vehicle.format()
    })



@api.route('vehicles', methods=['POST'])
def create_vehicle():
    data = request.json
    roles = ['plate_number', 'container_size', 'tank_size', 'employee_ssn']
    if not validate(roles, data):
        abort(400)
    plate_number = data['plate_number']
    container_size = data['container_size']
    tank_size = data['tank_size']
    employee_ssn = data['employee_ssn']
    driver = Employee.query.get(employee_ssn)
    if not driver:
        abort(404)
    # try:
    vehicle = Vehicle(plate_number=plate_number, container_size=container_size,
tank_size=tank_size, driver=driver)
    vehicle.save(True)
    return jsonify({
        "success": True,
        "vehicle": [vehicle.format()]
    })
    # except:
    #     abort(422)



@api.route('employees')
def get_employees():
    employees = Employee.query.all()
    employees_list = [employee.format() for employee in employees]
    return jsonify({
        "total_employees": len(employees_list),
        "employees": employees_list
    })


@api.route('employees/<int:employee_ssn>')
def get_employee(employee_ssn):
    employee = Employee.query.get(employee_ssn)
    return jsonify({
        "employee": employee.format()
    })


@api.route("employees", methods=['POST'])
def create_new_employee():
    data = request.json
    ssn = data['ssn']
    full_name = data['full_name']
    user_name = data['user_name']
    password = data['password']
    date_of_birth = data['data_of_birth']
    phone = data['phone']
    print(ssn)
    if not ssn or not full_name or not user_name or not password or not date_of_birth or not phone:
        abort(400)
    employee = Employee(SSN=ssn, full_name=full_name, user_name=user_name,
password=password, DOB=date_of_birth,
                        phone=phone).save(True)
    return jsonify({
        "success": True,
        # "employee": [employee.format()]
    })
```

```
@api.route("employees", methods=['PATCH'])
def update_supervisor():
    # TODO update the supervisor for all employee
    return ''


@api.route('employees/<int:employee_ssn>', methods=['DELETE'])
def delete_employee(employee_ssn):
    employee = Employee.query.get(employee_ssn)
    if employee is None:
        abort(404)
    employee.update()
    return jsonify({
        "success": True
    })


@api.route('users')
def get_all_users():
    users = User.query.all()
    users_list = [user.format() for user in users]
    return jsonify({"user": users_list})


@api.route('users', methods=['POST'])
def create_new_user():
    data = request.json
    user_name = data['user_name']
    first_name = data['first_name']
    last_name = data['last_name']
    email = data['email']
    password = data['password']
    gender = data['gender']
    area = data['area_code']
    area = Area.query.get(area)
    if not area:
        abort(404)
    roles = ['user_name', 'first_name', 'last_name', 'email', 'password', 'gender']
    abort(400) if not validate(roles, data) else None
    try:
        user = User(user_name=user_name, first_name=first_name, last_name=last_name, email=email,
password=password,
                    gender=gender, area=area).save(True)
        return jsonify({
            "success": True,
            'user': user.format()
        })
    except:
        abort(422)



@api.route('wastes')
def get_waste():
    data = request.args
    basket_id = data.get('basket_id', 0, int)
    wastes = Waste.query.all() if not basket_id else Waste.query.filter_by(basket_id=basket_id).all()
    wastes_list = [waste.format() for waste in wastes]
    total_size = 0
    for waste in wastes_list:
        total_size += +waste['size']

    return jsonify({
        "total_wastes_size": total_size,
        "wastes": wastes_list,
    })



@api.route('wastes', methods=['POST'])
def insert_new_waste():
    data = request.json
    basket = Basket.query.get(data['basket_id'])
    is_full = basket.set_wastes_height(data['waste_height'])
    if is_full:
        abort(422)
    waste_size = basket.get_waste_volume(data['waste_height'])
    waste = Waste(size=waste_size, type='bio', DOC=datetime.utcnow(), basket=basket).save()
    return jsonify({
        "basket_level": basket.get_basket_level(),
        "waste": waste.format()
    })



@api.route('wastes', methods=['DELETE'])
def delete_waste():
    # waste = Waste.query.filter_by(type='bio').delete()
    # db.session.commit()
    waste = Waste.query.all()
    print(waste)

    return ''



@api.route('test', methods=['POST', "GET"])
def test():
    data = request.json
    return jsonify({
        "value": data['value']
    })



@api.route("/baskets_types")
def get_basket_type():
    types_of_baskets = BasketType.query.all()
    type_list = [type_of_basket.format() for type_of_basket in types_of_baskets]
    return jsonify({
        "types": type_list
    })



@api.route("/baskets_types", methods=["POST"])
def create_basket_type():
    data = request.json
    length = data["length"]
    height = data["height"]
    width = data["width"]
    micro_controller = data["micro_controller"]
    roles = ['length', 'height', 'width']
    abort(400) if not validate(roles, data) else None
    try:
        basket_type = BasketType(length=length, height=height, width=width,
micro_controller=micro_controller).save()
        return jsonify({
            "success": True,
            'Type': basket_type.format()
        })
    except:
        abort(422)
```

```
@api.route('/baskets/<int:basket_id>/versions')
def get_basket_software_version(basket_id):
    basket = Basket.query.get(basket_id)
    software_versions =
SoftwareVersion.query.filter_by(basket_id=basket_id).order_by(SoftwareVersion.date.desc()).all()
    list_software_version = []
    status = 'update'
    for software_version in software_versions:
        if software_version.version == basket.software_version:
            status = 'rollback'
            list_software_version.append(software_version.format('current'))
            continue
        list_software_version.append(software_version.format(status))
    return jsonify({
        "software_versions": list_software_version,
        "current_version": basket.software_version
    })


@api.route("/software_versions/<string:version>")
def get_file(version):
    software = SoftwareVersion.query.get(version)
    file_name = "{}.bin".format(software.version)
    return send_file(BytesIO(software.file), attachment_filename=file_name, as_attachment=True)


@api.route("/software_versions", methods=["POST"])
def post_file():
    file = request.files['file']
    update_type = request.form.get("update_type", None)
    basket_id = request.form.get("basket_id", None)
    basket = Basket.query.get(basket_id)
    last_version = 
SoftwareVersion.query.filter_by(basket_id=basket_id).order_by(SoftwareVersion.date.desc()).first()
    if last_version:
        major, minor, patch = last_version.version.split(".")
        if update_type == "patch":
            patch = int(patch) + 1
        elif update_type == "minor":
            minor = int(minor) + 1
            patch = 0
        elif update_type == "major":
            major = int(major) + 1
            patch = 0
            minor = 0
        else:
            abort(422)
        print(last_version.version.split())
        version = "{}.{}.{}".format(major, minor, patch)
    else:
        version = "0.1.0"
    print(version)
    print(last_version)
    software_version = SoftwareVersion(version=version, file=file.read(), basket=basket)
    software_version.save(True)
    return jsonify({
        "success": True,
        "version": software_version.version
    })
```

APP→SITE→ROUTES.PY

```
from flask import Blueprint


site = Blueprint('site', __name__)
```

```
@site.route('/')
def index():
    return "<h1>Welcome to Our Waste Management System</h1>"



@site.route('/health')
def health_check():
    return "ok"
```

APP→VALIDATE→_INIT_.PY

```
def validate(roles, data):
    if roles is None:
        return True
    for role in roles:
        print(role)
        print(data)
        print(role in data)
        if role not in data:
            return False
    return True
```

APP→_INIT_.PY

```
from flask import Flask
from flask_cors import CORS
from app.site.routes import site
from app.api.routes import api
from app.models import setup_db



def create_app():
    app = Flask(__name__)
    if app.config["ENV"] == "production":
        app.config.from_object("config.ProductionConfig")
    else:
        app.config.from_object("config.DevelopmentConfig")
    CORS(app)
    app.register_blueprint(site)
    app.register_blueprint(api)
    setup_db(app)
    return app
```

APP→MODELS.PY

```
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
from datetime import datetime

db = SQLAlchemy()



def setup_db(app):
    db.app = app
    db.init_app(app)
    migrate = Migrate(app, db)



def commit():
    db.session.commit()



collect = db.Table('collect',
                   db.Column('plate_number', db.Integer, db.ForeignKey('vehicles.plate_number'),
primary_key=True),
                   db.Column('basket_id', db.Integer, db.ForeignKey('baskets.id'), primary_key=True),
                  db.Column('DOC', db.DateTime, primary_key=True)
                  )


complaint = db.Table('complaint',
                    db.Column('user_name', db.String, db.ForeignKey('users.user_name'),
primary_key=True),
                    db.Column('basket_id', db.Integer, db.ForeignKey('baskets.id'), primary_key=True),
                    db.Column('date_of_compliant', db.DateTime, primary_key=True),
                    db.Column('compliant_message', db.String),
                    )



class Basket(db.Model):
    __tablename__ = 'baskets'
    id = db.Column(db.Integer, primary_key=True)
    longitude = db.Column(db.Float, nullable=False)
    latitude = db.Column(db.Float, nullable=False)
    software_version = db.Column(db.String, nullable=False, default="0.0.0")
    wastes_height = db.Column(db.Integer, nullable=False, default=0)
    wastes = db.relationship('Waste', lazy=True, backref=db.backref('basket', lazy=True))
    software_versions = db.relationship('SoftwareVersion', lazy=True, backref=db.backref('basket',
lazy=True))
    type = db.Column(db.Integer, db.ForeignKey('basketsTypes.id'), nullable=False)
    area_code = db.Column(db.Integer, db.ForeignKey('areas.code'), nullable=False)


    def save(self):
        if self.id is None:
            db.session.add(self)
        db.session.commit()
        return self


    def delete(self):
        db.session.delete(self)
        db.session.commit()

    def format(self):
        return {
            "id": self.id,
            "longitude": self.longitude,
            "latitude": self.latitude,
            "software_version": self.software_version,
            "micro_controller": self.basketType.micro_controller,
            "level": "{}%".format(self.get_basket_level())
        }

    def set_wastes_height(self, waste_height):
        if waste_height <= (self.basketType.height - self.wastes_height):
            self.wastes_height += waste_height
            return False
        return True

    def get_waste_volume(self, height):
        return (self.length * self.width * float(height)) / 1000000

    def get_basket_level(self):
        return int((self.wastes_height / self.basketType.height) * 100)



class Area(db.Model):
    __tablename__ = "areas"
    code = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, unique=True, nullable=False)
    size = db.Column(db.Float, nullable=False)
    longitude = db.Column(db.String, nullable=False)
    latitude = db.Column(db.String, nullable=False)
    city = db.Column(db.String, nullable=False)
    baskets = db.relationship('Basket', lazy=False, backref=db.backref('area'))
    users = db.relationship('User', backref=db.backref('area'))

    def save(self, has_key_by_default=False):
        if self.code is None or has_key_by_default:
            db.session.add(self)
        db.session.commit()

    def format(self):
        return {
            "area_code": self.code,
            "area_name": self.name,
            "area_size": self.size,
            "longitude": self.longitude,
            "latitude": self.latitude,
            "city": self.city
        }


class User(db.Model):
    __tablename__ = 'users'
    user_name = db.Column(db.String, primary_key=True)
    first_name = db.Column(db.String, nullable=False)
    last_name = db.Column(db.String, nullable=False)
    email = db.Column(db.String, nullable=False, unique=True)
    password = db.Column(db.String, nullable=False)
    gender = db.Column(db.String, nullable=False)
    DOB = db.Column(db.DateTime)
    phone = db.Column(db.String)
    area_code = db.Column(db.Integer, db.ForeignKey('areas.code'), nullable=False)
    baskets = db.relationship('Basket', secondary=complaint, lazy=True,
backref=db.backref('complainants'))
```

```
    def save(self, has_key_by_default=False):
        if self.user_name is None or has_key_by_default:
            db.session.add(self)
        db.session.commit()
        return self

    def format(self):
        return {
            "user_name": self.user_name,
            "first_name": self.first_name,
            "last_name": self.last_name,
            "email": self.email,
            "gender": self.gender,
            "Date_of_birth": self.DOB
        }


class Employee(db.Model):
    __tablename__ = 'employees'
    SSN = db.Column(db.BigInteger, primary_key=True)
    full_name = db.Column(db.String, nullable=False)
    user_name = db.Column(db.String, nullable=False, unique=False)
    password = db.Column(db.String, nullable=False)
    DOB = db.Column(db.DateTime, nullable=False)
    phone = db.Column(db.String)
    vehicle = db.relationship('Vehicle', uselist=False, lazy="select", backref=db.backref('driver'))
    supervise = db.relationship("Employee")
    supervise_SSN = db.Column(db.BigInteger, db.ForeignKey('employees.SSN'), nullable=True)

    def save(self, has_key_by_default=False):
        if self.SSN is None or has_key_by_default:
            db.session.add(self)
        db.session.commit()

    def delete(self):
        db.session.delete(self)
        db.session.commit()

    def format(self):
        return {
            "SSN": self.SSN,
            "full_name": self.full_name,
            "user_name": self.user_name,
            "date_of_birth": self.DOB,
            "phone": self.phone
        }


class Vehicle(db.Model):
    __tablename__ = "vehicles"
    plate_number = db.Column(db.Integer, primary_key=True)
    container_size = db.Column(db.Float)
    tank_level = db.Column(db.Float)
    tank_size = db.Column(db.Float)
    employee_SSN = db.Column(db.BigInteger, db.ForeignKey('employees.SSN'))
    baskets = db.relationship('Basket', secondary=collect, lazy=True, backref=db.backref('baskets'))

    def save(self, has_key_by_default=False):
        if self.plate_number is None or has_key_by_default:
            db.session.add(self)
        db.session.commit()

    def format(self):
        return {
            "plate_number": self.plate_number,
            "container_size": self.container_size,
            "tank_level": self.tank_level,
            "tank_size": self.tank_size,
            "driver": self.driver.format() if self.driver else {}
        }


class Waste(db.Model):
    __tablename__ = "wastes"
    id = db.Column(db.Integer, primary_key=True)
    size = db.Column(db.Float)
    type = db.Column(db.String)
    DOC = db.Column(db.DateTime, nullable=False)
    basket_id = db.Column(db.Integer, db.ForeignKey('baskets.id'), nullable=True)

    def save(self, has_key_by_default=False):
        if self.id is None or has_key_by_default:
            db.session.add(self)
            db.session.add(self.basket)
        db.session.commit()
        return self

    def format(self):
        return {
            "basket_id": self.basket_id,
            "size": self.size,
            "type": self.type,
            "date_of_creation": self.DOC
        }
    def delete(self):
        db.session.commit()



class SoftwareVersion(db.Model):
    __tablename__ = "software_versions"
    version = db.Column(db.String(), primary_key=True)
    file = db.Column(db.LargeBinary())
    date = db.Column(db.DateTime, server_default=db.func.now())
    basket_id = db.Column(db.Integer, db.ForeignKey('baskets.id'), nullable=True, primary_key=True)

    def save(self, has_key_by_default=False):
        if self.version is None or has_key_by_default:
            db.session.add(self)
        db.session.commit()
        return self

    def format(self, status):
        return {
            "version": self.version,
            "date": self.date,
            "status": status
        }

    def delete(self):
        db.session.commit()



class BasketType(db.Model):
    __tablename__ = "basketsTypes"
    id = db.Column(db.Integer, primary_key=True)
    length = db.Column(db.Integer, nullable=False)
    width = db.Column(db.Integer, nullable=False)
    height = db.Column(db.Integer, nullable=False)
    micro_controller = db.Column(db.String, nullable=False)
    basket = db.relationship('Basket', lazy=True, backref=db.backref('basketType', lazy=True))

    def save(self):
        if self.id is None:
            db.session.add(self)
        db.session.commit()
        return self

    def format(self):
        return {
            "name": "{}*{}*{}/{}".format(self.length, self.width, self.height, self.micro_controller),
            "value": self.id
        }
```

MIGRATIONS→ALEMBICNIC.INI

```
#  A generic, single database configuration.

[alembic]
# template used to generate migration files
# file_template = %%(rev)s_%%(slug)s

# set to 'true' to run the environment during
# the 'revision' command, regardless of autogenerate
# revision_environment = false



# Logging configuration
[loggers]
keys = root,sqlalchemy,alembic
```

```
[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = WARN
handlers = console
qualname =

[logger_sqlalchemy]
level = WARN
handlers =
qualname = sqlalchemy.engine

[logger_alembic]
level = INFO
handlers =
qualname = alembic

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = generic

[formatter_generic]
format = %(levelname)-5.5s [%(name)s] %(message)s
datefmt = %H:%M:%S
```

MIGRATIONS→ENV.PY

```python
from __future__ import with_statement

import logging
from logging.config import fileConfig

from sqlalchemy import engine_from_config
from sqlalchemy import pool
from flask import current_app

from alembic import context

# this is the Alembic Config object, which provides
# access to the values within the .ini file in use.
config = context.config

# Interpret the config file for Python logging.
# This line sets up loggers basically.
fileConfig(config.config_file_name)
logger = logging.getLogger('alembic.env')

# add your model's MetaData object here
# for 'autogenerate' support
# from myapp import mymodel
# target_metadata = mymodel.Base.metadata
config.set_main_option(
    'sqlalchemy.url',
    str(current_app.extensions['migrate'].db.engine.url).replace('%', '%%'))
target_metadata = current_app.extensions['migrate'].db.metadata

# other values from the config, defined by the needs of env.py,
# can be acquired:
# my_important_option = config.get_main_option("my_important_option")
# ... etc.


def run_migrations_offline():
    """Run migrations in 'offline' mode.

    This configures the context with just a URL
    and not an Engine, though an Engine is acceptable
    here as well.  By skipping the Engine creation
    we don't even need a DBAPI to be available.

    Calls to context.execute() here emit the given string to the
    script output.

    """
    url = config.get_main_option("sqlalchemy.url")
    context.configure(
        url=url, target_metadata=target_metadata, literal_binds=True
    )

    with context.begin_transaction():
        context.run_migrations()


def run_migrations_online():
    """Run migrations in 'online' mode.

    In this scenario we need to create an Engine
    and associate a connection with the context.

    """
    # this callback is used to prevent an auto-migration from being generated
    # when there are no changes to the schema
    # reference: http://alembic.zzzcomputing.com/en/latest/cookbook.html
    def process_revision_directives(context, revision, directives):
        if getattr(config.cmd_opts, 'autogenerate', False):
            script = directives[0]
            if script.upgrade_ops.is_empty():
                directives[:] = []
                logger.info('No changes in schema detected.')

    connectable = engine_from_config(
        config.get_section(config.config_ini_section),
        prefix='sqlalchemy.',
        poolclass=pool.NullPool,
    )

    with connectable.connect() as connection:
        context.configure(
            connection=connection,
            target_metadata=target_metadata,
            process_revision_directives=process_revision_directives,
            **current_app.extensions['migrate'].configure_args
        )

        with context.begin_transaction():
            context.run_migrations()


if context.is_offline_mode():
    run_migrations_offline()
else:
    run_migrations_online()
```

MIGRATIONS→SCRIPT.PY.MAKO

```
"""${message}


Revision ID: ${up_revision}
Revises: ${down_revision | comma,n}
Create Date: ${create_date}


"""
from alembic import op
import sqlalchemy as sa
${imports if imports else ""}


# revision identifiers, used by Alembic.
revision = ${repr(up_revision)}
down_revision = ${repr(down_revision)}
branch_labels = ${repr(branch_labels)}
depends_on = ${repr(depends_on)}



def upgrade():
    ${upgrades if upgrades else "pass"}



def downgrade():
    ${downgrades if downgrades else "pass"}
```

CONFIG.PY

```
import os
from dotenv import load_dotenv
```

```
load_dotenv()


CONFIG_PATH = os.path.abspath(__file__)
ROOT_DIR = os.path.dirname(CONFIG_PATH)
UPLOAD_DIRECTORY = os.path.join(ROOT_DIR, 'uploads')



class Config(object):
    DEBUG = False
    TESTING = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SQLALCHEMY_DATABASE_URI = os.environ.get("DATABASE_URL")
    SESSION_COOKIE_SECURE = True



class ProductionConfig(Config):
    DEBUG = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False



class DevelopmentConfig(Config):
    ENV = "development"
    DEVELOPMENT = True
    SQLALCHEMY_TRACK_MODIFICATIONS = True
```

REQUIREMENTS

```
alembic==1.5.2
click==7.1.2
Flask==1.1.2
Flask-Cors==3.0.10
Flask-Migrate==2.6.0
Flask-SQLAlchemy==2.4.4
```

```
greenlet==1.0.0
gunicorn==20.0.4
importlib-metadata==3.10.0
itsdangerous==1.1.0
Jinja2==2.11.2
Mako==1.1.4
MarkupSafe==1.1.1
psycopg2==2.8.6
python-dateutil==2.8.1
python-dotenv==0.16.0
python-editor==1.0.4
six==1.15.0
SQLAlchemy==1.3.20
typing-extensions==3.7.4.3
Werkzeug==1.0.1
yapf==0.30.0
zipp==3.4.1
```

WSGI.PY

```
from app import create_app
app = create_app()
```