Usando Flask en el mundo real

Extensiones y buenas prácticas

PYCONES
ALICANTE 2019

Un poco sobre mí, muy poco

```python
class j2logo(WebDeveloper, Pythonista):
    def __init__(self):
        self.name = 'Juan José'
        self.surname = 'Lozano Gómez'
        self.personal_site = 'j2logo.com'
        self.email = 'juanjo@j2logo.com'
        self.degree = 'Ingeniero Informático'
        self.company = 'Answare Tech <answare-tech.com>'
        self.post = 'CTO'

    def get_fullname(self):
        return f'{self.name} {self.surname}'

    def social_networks(self):
        return {'Facebook': 'j2logo', 'Twitter': '@j2logo'}
```

OBJETIVO

Dar las claves para crear aplicaciones complejas de todo tipo con Flask

Flask - ¿El Micro framework?

Lo que vemos en todas partes

Flask es un framework **minimalista** escrito en Python que permite crear aplicaciones web **rápidamente**, de **forma sencilla** y con un **mínimo número de líneas** de código

Una definición mejor

Flask es framework web basado en WSGI. Está diseñado para que comenzar sea rápido y fácil, con la capacidad de escalar a aplicaciones complejas. Comenzó como una simple envoltura alrededor de Werkzeug y Jinja.

Flask ofrece sugerencias, pero no impone ninguna dependencia o diseño del proyecto. Depende del desarrollador elegir las herramientas y bibliotecas que desea usar.

Aplicaciones del mundo real

Aplicaciones del mundo real

Cosas deseables de un framework:

- Estructura aplicación
- Depuración
- Test
- Enrutamiento
- Extensibilidad
- Sesiones
- Cookies
- Acceso a los parámetros de la petición
- Gestión de errores
- Logs

Elementos comunes en proyectos web

- API
- Web + API
- AJAX / JSON
- Gestión usuarios
- Manejo de tokens (JWT)
- Autorización
- Envío de correo
- Notificaciones push
- Base de datos relacional
- Base de datos NoSQL
- Migraciones de base de datos
- Websockets
- Tareas en background

Todo lo anterior y mucho más es posible con Flask

😃

Internet está lleno de "HolaMundo(s)"

Internet está lleno de "HolaMundo(s)"

```
# app.py

$ pip install flask


from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'



$ export FLASK_APP=hello.py
$ flask run
 * Running on
http://127.0.0.1:5000/
```

Con esto ya sabes Flask y puedes crear aplicaciones muy sencillas

😯😟😲

Extendiendo la funcionalidad base de Flask

Versatilidad

Es bueno aprender y conocer Flask porque con él puedes hacer lo que quieras

Webs Backends APIs Microservicios ...

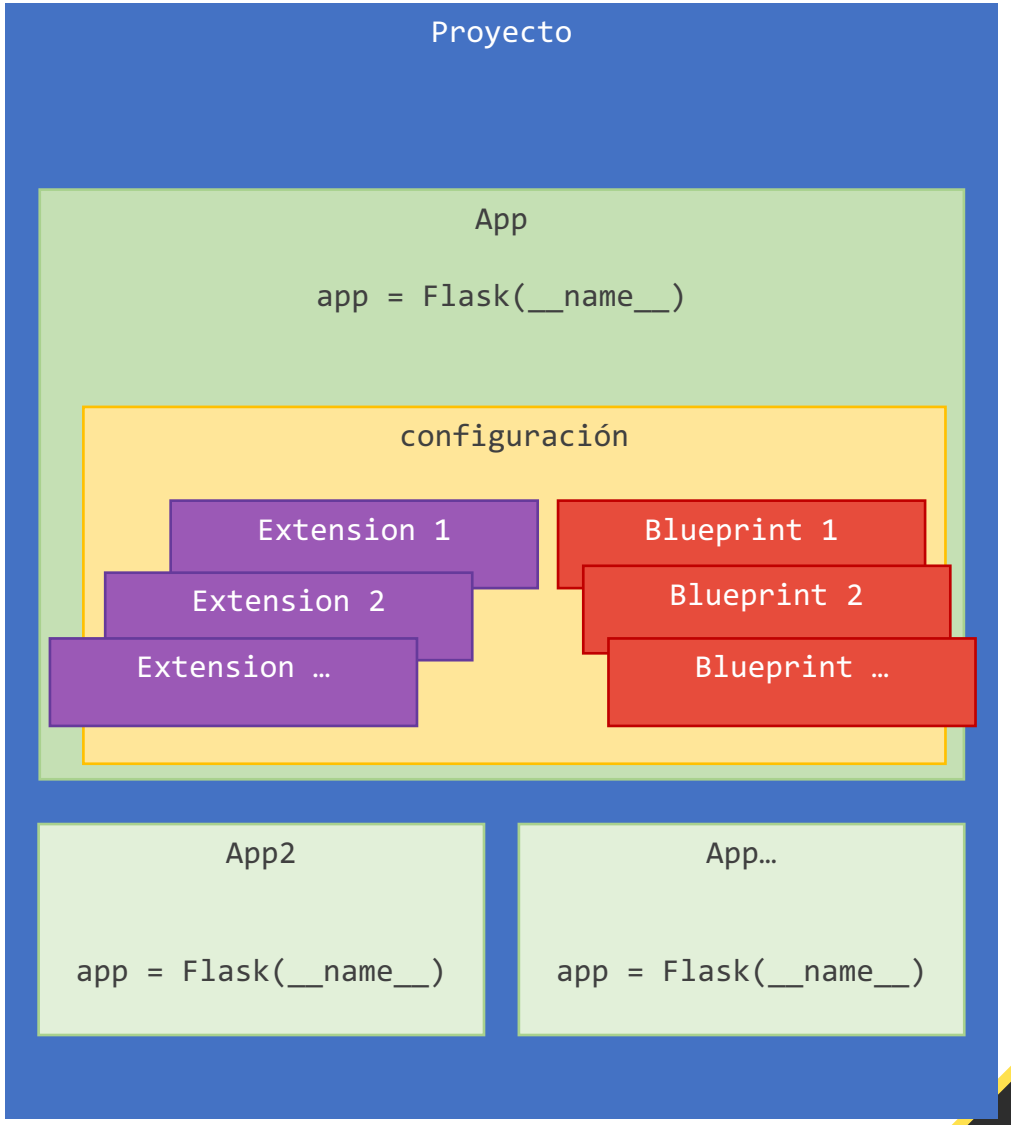
! Recuerda:

- Capacidad para escalar a aplicaciones complejas
- NO impone dependencias
- NO define una estructura de proyecto

Puntos de extensión

Blueprints

- Similares a un objeto aplicación (Flask) pero NO lo son (Blueprint)
- Registrar nuevas operaciones en la aplicación (views), múltiples veces, con diferentes URLS (también templates, static files y template filters)
- Opcionalmente pueden tener un prefijo de URL



Mis principales extensiones

- Flask-Babel: Internacionalización y localización
- Flask-Migrate: Migraciones de bases de datos con SQLAlchemy
- Flask-SocketIO: Websockets y comunicación en tiempo real
- Flask-SQLAlchemy: ORM base de datos relacionales
- Flask-WTF: Formularios web
- Flask-Cors: Aceptar peticiones desde otros dominios
- Flask-JWT-Extended: Manejar tokens JWT
- Flask-Mail: Envío de correo electrónico
- flask-marshmallow: Serializar/deserializar objetos JSON
- Flask-RESTful: APIs
- flask-swagger-ui: Mostrar la documentación del API
- Flask-Login: Utilidades para autenticación basada en cookies
- Flask-mongoengine: ORM mongo

Más extensiones en

https://github.com/humiaozuzu/awesome-flask

Necesitamos una estructura base de aplicación

Proyecto muy sencillo

```
+_mi-proyecto
|_app.py
|_+_static/
|_+_templates/
```

```
+_mi-proyecto
|_app.py
|_forms.py
|_models.py
|_+_static/
|_+_templates/
```

Necesitamos una estructura de aplicación

Estructura de proyecto

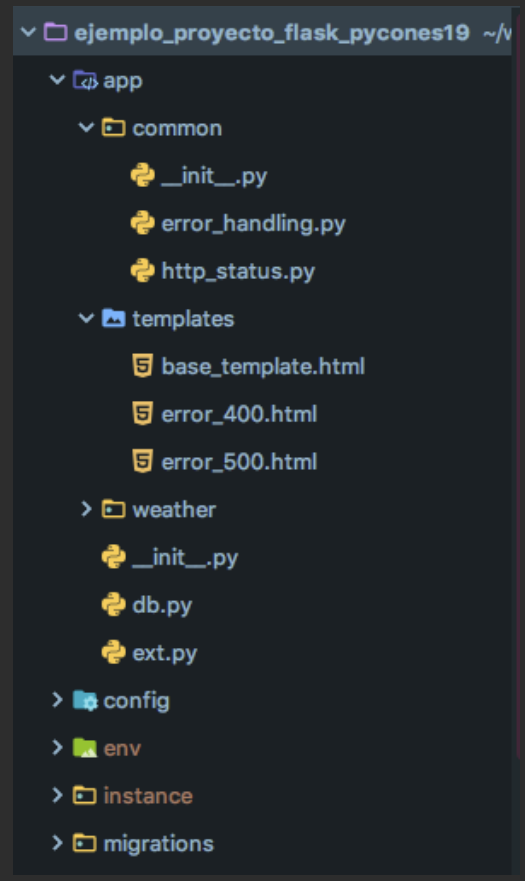
```
+ mi_proyecto/
   |_ app/
      |_ __init__.py
      |_ db.py
      |_ ext.py
      |_ common/
      |_ mod1/
         |_ __init__.py
         |_ resources.py
         |_ routes.py
         |_ templates/
            |_ mod1/
               |_ template1.html
               |_ template2.html
         |_ models.py
         |_ forms.py
         |_ ...
      |_ mod2/
         |_ ...
      |_ ...
      |_ static/
         |_ css/
         |_ images/
         |_ js/
      |_ templates/
         |_ base_template.html
         |_ ...
   |_ config
      |_ default.py
      |_ development.py
      |_ local.py
      |_ production.py
      |_ testing.py
   |_ env/
   |_ fixtures/
   |_ instance
      |_ __init__.py
      |_ config.py
   |_ .gitignore
   |_ CHANGELOG.md
   |_ entrypoint.py
   |_ README.md
   |_ requirements.txt
```

División estructural VS División funcional

- __init__.py: Métodos factoría y configuración
- db.py: Referencia al objeto de sesión de BD y modelos base
- ext.py: Declaración e instancias de las extensiones
- common/: Módulos comunes (helpers, utils, …)
- modXX/: Paquete con una funcionalidad específica de la app
  - resources.py: Endpoints del API
  - routes.py: Vistas
  - templates/: Plantillas específicas del módulo
  - models.py: Modelos
  - forms.py: Formularios
  - …
- static/: Recursos estáticos
- templates/: Plantillas html
- config/: Ficheros de configuración, uno por entorno
- env/: Entorno virtual con las bibliotecas externas al proyecto
- fixtures/: Ficheros y recursos de utilidad
- instance/: Configuración local fuera del control de versiones
- entrypoint.py: Instanciación de las apps del proyecto, al menos una
- requirements.txt: Declaración de dependencias externas al proyecto

Creando una aplicación del mundo real

Ejemplo de aplicación del tiempo que captura la temperatura de una estación meteorológica 1 vez a la hora y se la muestra a un usuario + API para aplicación móvil



https://github.com/j2logo/proyecto_flask_base

app/db.py

Base de datos relacional (PostgreSQL)

```python
from flask_sqlalchemy import SQLAlchemy

from app.common.error_handling import ObjectNotFound

db = SQLAlchemy()


class BaseModelMixin:

    def save(self):
        db.session.add(self)
        db.session.commit()

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

    @classmethod
    def get_all(cls):
        return cls.query.all()
```

Objeto global SQLAlchemy

Mixin Base del que heredan todos los modelos con métodos comunes

app/ext.py

```python
from celery import Celery
from flask_marshmallow import Marshmallow
from flask_migrate import Migrate

ma = Marshmallow()
migrate = Migrate()
celery = Celery('flask_PyConES19')
```

Objetos globales con una referencia a las extensiones usadas en el proyecto

Con este fichero se evitan las dependencias circulares

Creando una aplicación del mundo real

app/weather/models.py

```python
import datetime
from sqlalchemy import and_
from app.db import db, BaseModelMixin

class TempRecord(db.Model, BaseModelMixin):

    id = db.Column(db.Integer, primary_key=True)
    fetched = db.Column(db.DateTime, nullable=False)
    temp = db.Column(db.Float, nullable=False)

    def __init__(self, temp, fetched=None):
        self.temp = temp
        if fetched is None:
            self.fetched = datetime.datetime.utcnow()
        else:
            self.fetched = fetched

    def __str__(self):
        return f'Temp: {self.temp} | Fetched: {self.fetched}'

    def __repr__(self):
        return self.__str__()

    @staticmethod
    def get_records_between_dates(start, end):
        return TempRecord.query.filter(and_(TempRecord.fetched >= start,
TempRecord.fetched < end)).all()
```

app/weather/schemas.py

```python
from marshmallow import fields

from app.ext import ma


class TempRecordSchema(ma.Schema):
    id = fields.Integer(dump_only=True)
    fetched = fields.DateTime()
    temp = fields.Float()
```

Para serializar el modelo en JSON

app/weather/tasks.py

```python
import datetime
import random

from app.ext import celery
from app.weather.models import TempRecord


@celery.task()
def fetch_temp():
    """
    Esta función es un fake que simula la captura de temperatura de un sensor
    generando un número aleatorio
    """
    rand_init = random.SystemRandom()
    random_temp = round(rand_init.uniform(0, 45), 1)
    temp_record = TempRecord(random_temp, datetime.datetime.utcnow())
    temp_record.save()
```

app/weather/helpers.py

```python
import datetime

from app.weather.exceptions import NoTempsForTodayException
from app.weather.models import TempRecord


def average_temperature_of_today():
    today = datetime.datetime.utcnow().date()
    today_dt = datetime.datetime(today.year, today.month, today.day, 0, 0)
    tomorrow_dt = today_dt + datetime.timedelta(days=1)
    today_records = TempRecord.get_records_between_dates(today_dt, tomorrow_dt)
    if len(today_records) == 0:
        raise NoTempsForTodayException('No se han registrado temperaturas en el día de hoy')
    temp_add = 0
    for r in today_records:
        temp_add += r.temp
    return round(temp_add/len(today_records), 1)
```

Clave para reutilizar en la web y en el API

Creando una aplicación del mundo real

app/weather/routes.py

```python
import logging

from flask import Blueprint, render_template

from app.weather.exceptions import CustomWeatherException, NoTempsForTodayException
from app.weather.helpers import average_temperature_of_today
from app.weather.models import TempRecord

logger = logging.getLogger(__name__)

weather_bp = Blueprint('weather_bp', __name__, template_folder='templates')


@weather_bp.route("/", methods=['GET'])
def index():
    records = TempRecord.get_all()
    template_context = dict()
    template_context['records'] = records
    template_context['title'] = 'Listado de temperaturas - Flask PyConES19'
    return render_template("weather/index.html", **template_context)


@weather_bp.route("/temp/today/avg", methods=['GET'])
def average_temp():
    template_context = dict()
    try:
        avg = average_temperature_of_today()
        template_context['avg'] = avg
    except NoTempsForTodayException:
        template_context['error'] = 'No hay temperaturas'
    template_context['title'] = 'Temperatura media de hoy - Flask PyConES19'
    return render_template("weather/average.html", **template_context)
```

app/weather/resources.py

```python
import logging
from flask import Blueprint, jsonify, current_app
from flask_restful import Api, Resource
from app.weather.exceptions import CustomWeatherException, NoTempsForTodayException
from app.weather.helpers import average_temperature_of_today
from app.weather.models import TempRecord
from app.weather.schemas import TempRecordSchema

logger = logging.getLogger(__name__)
weather_api_bp = Blueprint('temp_api_bp', __name__)
api = Api(weather_api_bp)
temp_record_schema = TempRecordSchema()


class TempListResource(Resource):
    def get(self):
        logger.info('Obteniendo el listado de temperaturas')
        current_app.logger.info('Obteniendo el listado de temperaturas')
        records = TempRecord.get_all()
        response = temp_record_schema.dump(records, many=True)
        return response


class AvgTempResource(Resource):
    def get(self):
        try:
            avg = average_temperature_of_today()
            return jsonify({'avg': avg})
        except NoTempsForTodayException:
            return jsonify({'error': 'No hay temperaturas'})


api.add_resource(TempListResource, '/api/temp', endpoint='temp_list_resource')
api.add_resource(AvgTempResource, '/api/temp/today/avg', endpoint='avgtemp_resource')
```

Creando una aplicación del mundo real

Gestión de errores

app/weather/exceptions.py

```python
class WeatherException(Exception):
    pass


class CustomWeatherException(WeatherException):
    pass


class NoTempsForTodayException(WeatherException):
    pass
```

app/weather/__init__.py

```python
def init_app(app):
    _set_error_handlers_callbacks(app)


def _set_error_handlers_callbacks(app):

    @app.errorhandler(WeatherException)
    def handle_weather_exception(e):
        return weather_error_callback(str(e))

    @app.errorhandler(CustomWeatherException)
    def handle_custom_weather_exception(e):
        return custom_error_callback(str(e))
```

app/weather/error_callbacks.py

```python
from flask import jsonify, render_template, request


def weather_error_callback(error_string):
    if request.path.startswith('/api'):
        return jsonify({'msg': error_string}), 500
    else:
        return render_template('error_500.html'), 500


def custom_error_callback(error_string):
    if request.path.startswith('/api'):
        return jsonify({'msg': error_string}), 400
    else:
        return render_template('error_400.html'), 400
```

Gestión de errores

app/weather/routes.py

```
@weather_bp.route('/error/<int:num>', methods=['GET'])
def show_error(num):
    if num == 10:
        raise CustomWeatherException('Has introducido el número 10')
    else:
        return index()
```

Creando una aplicación del mundo real

app/__init__.py

```python
def create_app(settings_module):
    app = Flask(__name__, instance_relative_config=True)
    # Load the config file specified by the APP environment variable
    app.config.from_object(settings_module)
    # Load the configuration from the instance folder
    if not app.config.get('TESTING', False):
        app.config.from_pyfile('config.py', silent=True)
    else:
        app.config.from_pyfile('config-testing.py', silent=True)

    # Init third party modules
    db.init_app(app)
    ma.init_app(app)
    migrate.init_app(app, db)
    init_celery(app)

    # Init custom modules
    weather.init_app(app)

    # Blueprints registration
    app.register_blueprint(weather_bp)
    app.register_blueprint(weather_api_bp)

    # Custom error handlers
    register_error_handlers(app)

    return app
```

entrypoint.py

```python
import os

from flask import jsonify

from app import create_app

settings_module = os.getenv('APP_SETTINGS_MODULE')
app = create_app(settings_module)
```

¡Muchas GRACIAS!

Más en j2logo.com

juanjo@j2logo.com
Twitter: @j2logo | #python_es
Facebook: j2logo