Aquí te explico cuáles son los pasos principales que debes seguir para crear un API REST con Flask, te doy un ejemplo de estructura de aplicación y te muestro las posibles extensiones que puedes utilizar para que te hagan la vida más fácil.

A diferencia de una aplicación o página web, cuyos elementos principales son las páginas, el core de una API REST son los recursos. Un recurso es cualquier tipo de objeto, dato o servicio al que puede acceder un cliente. Los recursos se hacen accesibles a través de la interfaz del API, en la que se exponen los métodos y las URLs disponibles para acceder y/o manipular cada uno de los recursos.

En este tutorial vamos a crear un API con Flask para gestionar un catálogo de películas almacenado en una base de datos *Sqlite*. El API podrá ser consumido posteriormente por una app móvil, un frontend desarrollado en Javascript, un servicio, etc.

¿Por qué un API REST con Flask?

Como ya sabrás, Flask es un framework para desarrollo web escrito en Python. Se puede utilizar para diversos tipos de aplicación, entre ellas, desarrollo de APIs. De hecho, es uno de los mejores frameworks de desarrollo para implementar este tipo de soluciones por lo sencillo que resulta y las facilidades que ofrece. Yo es el framework que siempre utilizo en mis desarrollos web y APIs

Existen muchas maneras de implementar un API REST en Flask. Desde usar el framework con lo que ofrece de base, hasta instalar varias extensiones con diferentes configuraciones.

En este tutorial te voy a explicar cómo implementar un API REST en Flask utilizando las extensiones *Flask-Restful* y *Flask-Marshmallow* (basada en *Marshmallow*). El motivo de usar estas extensiones es que hacen el API mucho más robusta y más fácil de evolucionar cuando esta crece en complejidad.

Al contrario que en otros tutoriales que te encuentres por ahí, voy a empezar el proyecto al revés. En primer lugar definiremos la estructura completa de la aplicación. A continuación implementaremos los modelos de la base de datos y los

esquemas para serializar dichos modelos a JSON. Finalmente escribiremos el código asociado a los recursos y el código que unirá todo.

Recursos

Sin embargo, antes de dar paso al tutorial vamos a repasar qué es un recurso en un API REST.

Un recurso es un objeto con un tipo, datos asociados, relaciones con otros recursos y un conjunto de métodos que operan en él. Es similar a una instancia de objeto en un lenguaje de programación orientado a objetos, con la importante diferencia de que solo se definen unos pocos métodos estándar para el recurso (correspondientes a los métodos HTTP GET, POST, PUT y DELETE).

Los recursos se pueden agrupar, o no, en colecciones. Las colecciones son homogéneas, es decir, contienen un solo tipo de recurso y están desordenadas. En sí mismas, las colecciones también son recursos.

Por tanto, un API REST no es más que una interfaz para interactuar con los recursos que hay tras ella.

Y una vez que tienes claro qué es un recurso, ¡comenzamos!

Estructura de la aplicación

Para la aplicación vamos a seguir una estructura muy similar a la que se indica en el tutorial Estructura de un proyecto Flask.

Comencemos creando un directorio para el proyecto llamado `api-peliculas`. Accede al directorio, crea un entorno virtual y actívalo.

Una vez activado, debes crear una estructura de aplicación como la siguiente:

```
+api-peliculas
|_+ app
  |_+ common
    |_ __init__.py
    |_ error_handling.py   # Utilidades para el manejo de errores
  |_+ films
    |_+ api_v1_0
      |_ __init__.py
      |_ resources.py   # Endpoints del API
      |_ schemas.py     # Esquemas para serializar los modelos
    |_ __init__.py
    |_ models.py   # Modelos
  |_ __init__.py   # Configuración de la aplicación
  |_ db.py        # Configuración de la base de datos
  |_ ext.py       # Instanciación de las extensiones
|_+ config        # Directorio para la configuración
  |_ __init__.py
  |_ default.py    # Configuración por defecto
|_ entrypoint.py   # Crea la instancia de la app
```

Crea los paquetes y módulos que se indican de manera que el proyecto sea similar al ejemplo de arriba. Iremos dando contenido y explicando cada uno de los módulos a lo largo del tutorial.

Para crear el API, además de *Flask*, haremos uso de las siguientes extensiones:

- **Flask Restful**: Es una extensión que permite generar APIs REST muy fácilmente. Además, viene con un montón de utilidades.
- **Flask SQLAlchemy**: Para interactuar con la base de datos a través de su ORM. Puedes encontrar más información aquí.
- **Flask Migrate**: Esta extensión permite generar las tablas de la base de datos a partir de ficheros de migración. Puedes encontrar más información aquí.
- **Flask Marshmallow**: Es una extensión que facilita la serialización de los modelos de la base de datos a JSON y viceversa. Está basada en *Marshmallow*.
- **Marshmallow SQLAlchemy**: Para integrar *Flask Marshmallow* con *SQLAlchemy*.

Ejecuta el siguiente comando para instalar todas estas extensiones y sus dependencias:

```
$> pip install flask flask_restful flask_sqlalchemy flask_migrate flask_marshmallow marshmallow-sqlalchemy
```

A continuación, abre el fichero `ext.py` y añade lo siguiente:

```
from flask_marshmallow import Marshmallow
from flask_migrate import Migrate

ma = Marshmallow()
migrate = Migrate()
```

Este fichero lo utilizaremos para instanciar las distintas extensiones que utilicemos en la aplicación. En este caso, *Flask-Marshmallow* y *Flask-Migrate*. El hacerlo aquí evita que se produzcan referencias circulares cuando se utilicen a lo largo de la aplicación.

Te habrás fijado que no hemos instanciado *Flask-SQLAlchemy*. Esto lo haremos en el módulo `db.py`. Ábrelo y pega el siguiente código:

```
from flask_sqlalchemy import SQLAlchemy

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()

    @classmethod
    def get_by_id(cls, id):
        return cls.query.get(id)

    @classmethod
    def simple_filter(cls, **kwargs):
        return cls.query.filter_by(**kwargs).all()
```

Esta vez sí instanciamos `Flask-SQLAlchemy` en la variable `db`. Del mismo modo, lo hacemos aquí para evitar referencias circulares. Además, se crea la clase `BaseModelMixin` con métodos de utilidad para los modelos.

Los modelos

A continuación, vamos a crear los modelos del API. Como te indicaba al comienzo del tutorial, el API se utilizará para gestionar un catálogo de películas.

Vamos a utilizar dos modelos: `Film` y `Actor`. El modelo `Film` representa a una película del catálogo y está compuesto por los siguientes atributos:

- `id`, clave primaria.
- `title`, título de la película.
- `length`, duración (en segundos) de la película.
- `year`, año de estreno.
- `director`, director de la película.
- `actors`, lista con los actores de la película.

Por su parte, el modelo `Actor` representa a un actor y está formado por los siguientes atributos:

- `id`, clave primaria.
- `name`, nombre del actor.
- `film_id`, clave ajena a la película en la que aparece.

Abre el fichero `models.py` y guárdalo con el siguiente contenido:

```
from app.db import db, BaseModelMixin


class Film(db.Model, BaseModelMixin):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String)
    length = db.Column(db.Integer)
```

```
    year = db.Column(db.Integer)
    director = db.Column(db.String)
    actors = db.relationship('Actor', backref='film', lazy=False, cascade='all, delete-orphan')

    def __init__(self, title, length, year, director, actors=[]):
        self.title = title
        self.length = length
        self.year = year
        self.director = director
        self.actors = actors

    def __repr__(self):
        return f'Film({self.title})'

    def __str__(self):
        return f'{self.title}'


class Actor(db.Model, BaseModelMixin):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    film_id = db.Column(db.Integer, db.ForeignKey('film.id'), nullable=False)

    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f'Actor({self.name})'

    def __str__(self):
        return f'{self.name}'
```

Fíjate que del módulo `app/db.py` se importan las referencias a `db` y `BaseModelMixin`.

Los esquemas

Una vez que hemos creado la instancias de las extensiones y definido los modelos, el siguiente paso consiste en crear los esquemas. Para definir los esquemas utilizaremos Flask-Marshmallow. Un esquema es una clase que define cómo se serializa un modelo/recurso consumido por el API a JSON.

Un esquema también permite crear un modelo a partir de un JSON, pero esto no lo veremos en este tutorial.

Los esquemas se definen de manera muy similar a los modelos, solo que, para los campos, hay que utilizar los tipos definidos por *Marhsmallow*.

Además, para que la serialización se realice de forma automática, los nombres de cada uno de los campos del esquema deben ser idénticos a los nombres de los atributos del modelo que representan. Cuando *Marshmallow* encuentre un campo en el esquema que coincida con el nombre de un atributo en el modelo, tratará de serializarlo. Fíjate también de que los campos sean de un tipo compatible.

Abre el fichero `schemas.py` y pega el siguiente código:

```
from marshmallow import fields

from app.ext import ma


class FilmSchema(ma.Schema):
    id = fields.Integer(dump_only=True)
    title = fields.String()
    length = fields.Integer()
    year = fields.Integer()
    director = fields.String()
    actors = fields.Nested('ActorSchema', many=True)


class ActorSchema(ma.Schema):
    id = fields.Integer(dump_only=True)
    name = fields.String()
```

Como puedes ver hemos creado dos esquemas. Uno para el modelo `Film` y otro para el modelo `Actor`.

Observa que al campo `id` se le pasa como argumento `dump_only=True`. Esto hará que solo se tenga en cuenta este campo a la hora de serializar el objeto (y no en la carga). Por otro lado, observa que el campo `actors` en el esquema `FilmSchema` se define como `Nested`. Esto indica que el esquema `FilmSchema` contiene varios (`many=True`) elementos de tipo `ActorSchema`.

Los recursos

Lo último que queda por implementar en referencia al API son los recursos. Como hemos visto en la introducción, 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/PATCH**: Para modificar un recurso.
- **DELETE**: Para eliminar un recurso.

Para implementar los recursos en *Flask* haremos uso de la extensión *Flask-Restful*. En *Flask-Restful* 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.

Veámoslo en acción. En nuestro caso vamos a crear dos recursos diferentes:

- **Colección de películas**: Este recurso es en realidad una colección. Permitirá obtener el catálogo completo de películas y añadir una nueva al catálogo.
- **Película**: Este recurso obtendrá una película del catálogo a partir de su `id`.

Abre el fichero `resources.py` y pega el siguiente código:

```
from flask import request, Blueprint
from flask_restful import Api, Resource

from .schemas import FilmSchema
from ..models import Film, Actor

films_v1_0_bp = Blueprint('films_v1_0_bp', __name__)

film_schema = FilmSchema()

api = Api(films_v1_0_bp)

api.add_resource(FilmListResource, '/api/v1.0/films/', endpoint='film_list_resource')
api.add_resource(FilmResource, '/api/v1.0/films/<int:film_id>', endpoint='film_resource')
```

¿Qué hemos hecho en el código anterior?

- Definir un *Blueprint* llamado `films_v1_0_bp`. Siguiendo esta filosofía, podremos crear nuevos recursos en otros paquetes de la aplicación.

- Crear una instancia del `Api` a partir del blueprint anterior. Esta variable `api` es muy similar a un blueprint, solo que expone métodos propios de un `Api` de *Flask-Restful*.
- Crear una instancia del esquema `FilmSchema`.
- Definir el recurso `FilmListResource` asociado a la URL `/api/v1.0/films/`.
- Definir el recurso `FilmResource` asociado a la URL `/api/v1.0/films/<film_id>`. `film_id` será un parámetro que se defina en cada uno de los métodos del recurso.

Veamos ahora cómo implementar un recurso con *Flask-Restful*.

Recurso FilmListResource

Justo después de la línea `api=Api(films_v1_0_bp)` añade lo siguiente:

```
class FilmListResource(Resource):
    def get(self):
        films = Film.get_all()
        result = film_schema.dump(films, many=True)
        return result
```

Básicamente hemos creado la clase que define la colección `FilmListResource`. Todo recurso en *Flask-Restful* debe heredar de la clase `Resource`.

En principio, este recurso define el método `get()`. De momento, esto permitirá únicamente hacer peticiones `GET` a la URL `/api/v1.0/films/`.

Como puedes apreciar, lo único que hace este método es obtener en `films` el listado de objetos del modelo `Film`, serializarlos y devolver el JSON resultante. La serialización se realiza llamando al método `dump()` del esquema correspondiente. A este método se le pasa el objeto a serializar. En caso de que sea un listado, como ocurre aquí, hay que indicar el argumento `many=True`.

Ahora vamos a implementar el método `post()`. Este método se utilizará para añadir una nueva película a la colección. Justo a continuación del método `get()` añade lo siguiente:

```
def post(self):
    data = request.get_json()
    film_dict = film_schema.load(data)
    film = Film(title=film_dict['title'],
                length=film_dict['length'],
                year=film_dict['year'],
```

```
            director=film_dict['director']
    )
    for actor in film_dict['actors']:
        film.actors.append(Actor(actor['name']))
    film.save()
    resp = film_schema.dump(film)
    return resp, 201
```

Te explico paso a paso lo que hace el método `post()`:

- En primer lugar obtiene el cuerpo de la respuesta en formato JSON. Para ello, se llama al método `get_json()` del objeto `request`.
- A continuación se llama al método `load()` del esquema `film_schema`. Este método valida que el JSON `data` cumpla con el esquema. A su vez, se crea el diccionario `film_dict` a partir del JSON original.
- Se crea una instancia de `Film` a partir de los datos del diccionario y se guarda.
- Se serializa el objeto `film` y se devuelve en la respuesta. Fíjate que en esta ocasión también se indica el código de respuesta. En este caso `201`, que significa que se creó un objeto.

Recurso FilmResource

Este recurso es muy sencillo. Solo implementa el método `get()` que se utilizará para obtener una película a partir de su `id`. En esta ocasión, el método `get()` define el parámetro `film_id` que indicamos en el endpoint del recurso.

```
class FilmResource(Resource):
    def get(self, film_id):
        film = Film.get_by_id(film_id)
        resp = film_schema.dump(film)
        return resp
```

Control de errores de un API REST en Flask

Ahora vamos a ver cómo manejar errores del API. Imagina que haces una petición GET para obtener una película a partir de su `id` pero no existe. La aplicación mostraría un JSON vacío y el usuario no sabría que está ocurriendo.

En estos casos, es más lógico devolver al usuario un código de error `404` con un JSON explicativo.

Te muestro cómo hacerlo paso a paso.

En primer lugar abre el módulo `error_handling.py` y pega el siguiente código:

```
class AppErrorBaseClass(Exception):
    pass


class ObjectNotFound(AppErrorBaseClass):
    pass
```

En él se definen dos excepciones personalizadas. Una general, `AppErrorBaseClass`, de la que heredarán todas la excepciones y errores de la aplicación y `ObjectNotFound`, que utilizaremos cuando se intente acceder a un recurso que no existe.

A continuación, modifica el código del método `get()` del recurso `FilmResource` para que se lance la excepción `ObjectNotFound`.

En una sección posterior veremos dónde y cómo se maneja la excepción.

Configuración

Ya casi estamos terminando. En esta ocasión vamos a definir los parámetros de configuración de la aplicación tal y como se muestra en este tutorial.

Abre el fichero `config/default.py` y añade lo siguiente:

```
SECRET_KEY =
'123447a47f563e90fe2db0f56b1b17be62378e31b7cfd3adc776c59ca4c75e2fc512c15f69bb38307d11d5d17a41a79
36789'

PROPAGATE_EXCEPTIONS = True

# Database configuration
SQLALCHEMY_DATABASE_URI = 'sqlite:///films.sqlite'
SQLALCHEMY_TRACK_MODIFICATIONS = False
SHOW_SQLALCHEMY_LOG_MESSAGES = False

ERROR_404_HELP = False
```

Los parámetros de configuración son:

- `SECRET_KEY`: Lo utilizan Flask y ciertas extensiones que manejan aspectos de seguridad.
- `PROPAGATE_EXCEPTIONS`: Para propagar las excepciones y poder manejarlas a nivel de aplicación.
- `SQLALCHEMY_DATABASE_URI`: URI de la base de datos.
- `SQLALCHEMY_TRACK_MODIFICATIONS`: Se desactiva como indica la documentación.
- `SHOW_SQLALCHEMY_LOG_MESSAGES` = Se deshabilitan los mensajes de log de SQLAlchemy.
- `ERROR_404_HELP`: Deshabilita las sugerencias de otros endpoints relacionados con alguno que no exista (Flask-Restful).

La aplicación

En último lugar vamos a crear la aplicación y vamos a unir todo el código anterior para poner en marcha el API.

Abre el fichero `app/__init__.py` y pega el código siguiente:

```
from flask import Flask, jsonify
from flask_restful import Api

from app.common.error_handling import ObjectNotFound, AppErrorBaseClass
from app.db import db
from app.films.api_v1_0.resources import films_v1_0_bp
from .ext import ma, migrate


def create_app(settings_module):
    app = Flask(__name__)
    app.config.from_object(settings_module)

    # Inicializa las extensiones
    db.init_app(app)
    ma.init_app(app)
    migrate.init_app(app, db)

    # Captura todos los errores 404
    Api(app, catch_all_404s=True)

    # Deshabilita el modo estricto de acabado de una URL con /
    app.url_map.strict_slashes = False
```

```
    # Registra los blueprints
    app.register_blueprint(films_v1_0_bp)

    # Registra manejadores de errores personalizados
    register_error_handlers(app)

    return app


def register_error_handlers(app):
    @app.errorhandler(Exception)
    def handle_exception_error(e):
        return jsonify({'msg': 'Internal server error'}), 500

    @app.errorhandler(405)
    def handle_405_error(e):
        return jsonify({'msg': 'Method not allowed'}), 405

    @app.errorhandler(403)
    def handle_403_error(e):
        return jsonify({'msg': 'Forbidden error'}), 403

    @app.errorhandler(404)
    def handle_404_error(e):
        return jsonify({'msg': 'Not Found error'}), 404

    @app.errorhandler(AppErrorBaseClass)
    def handle_app_base_error(e):
        return jsonify({'msg': str(e)}), 500

    @app.errorhandler(ObjectNotFound)
    def handle_object_not_found_error(e):
        return jsonify({'msg': str(e)}), 404
```

En este módulo se implementa el método factoría que crea la `app` de Flask. Además se registran los manejadores de errores para los códigos de respuesta y excepciones no controladas. Fíjate cómo aquí se maneja la excepción `ObjectNotFound`, devolviendo un mensaje JSON con el error y el código de respuesta `404`.

Finalmente, abre el fichero `entrypoint.py` y pega el siguiente código:

Básicamente, en este módulo se crea la instancia de la `app` de Flask indicando dónde están definidos los parámetros de configuración (el fichero de configuración se especifica en la variable de entorno `APP_SETTINGS_MODULE`).

Pues ya casi podemos probar el API, pero antes hay que crear la base de datos y sus correspondientes tablas.

Además, para que la aplicación funcione y se pueda ejecutar, debemos crear las siguientes variables de entorno en el fichero `activate` del entorno virtual.

```
export FLASK_APP="entrypoint:app"
export FLASK_ENV="development"
export APP_SETTINGS_MODULE="config.default"
```

❗ **ATENCIÓN:** Si estás en un entorno Windows, el fichero se denomina `activate.bat` (si usas como terminal *cmd*) o `activate.ps1` (si usas como terminal *PowerShell*). Además, el modo de definir las variables de entorno varía al mostrado arriba.

Crear la base de datos y las tablas

Para crear la base de datos y las tablas, vamos a hacer uso de la extensión Flask-Migrate. Ejecuta lo siguientes comandos en el terminal:

```
$> flask db init
$> flask db migrate -m "Initial_db"
$> flask db upgrade
```

El API REST en Flask en funcionamiento

Ahora sí que podemos poner en marcha la aplicación y probar nuestro API. Para arrancar la aplicación ejecuta el siguiente comando desde el terminal:

```
$> flask run
```

Utiliza el programa que prefieras para hacer peticiones al API. Yo para este tutorial estoy utilizando *Postman*, pero tú puedes usar el que prefieras, o consumir el API de verdad utilizando código Javascript, Python, etc.

Añadir una película a la colección

Envía una petición `POST` a la URL `http://localhost:5000/api/v1.0/films/` con el siguiente contenido:

```
{
  "title": "Forrest Gump",
  "length": 8520,
  "year": 1994,
  "director": "Robert Zemeckis",
  "actors": [
    { "name": "Tom Hanks"},
    { "name": "Robin Wright"},
    { "name": "Gary Sinise"},
    { "name": "Mykelti Williamson"},
    { "name": "Sally Field"},
    { "name": "Michael Conner Humphreys"}
  ]
}
```


Obtener la colección de películas

Realiza una petición `GET` a la URL `http://localhost:5000/api/v1.0/films/`. La respuesta será similar a la siguiente:


```
[
  {
    "year": 1994,
    "actors": [
      {
        "name": "Gary Sinise",
        "id": 3
      },
      {
        "name": "Michael Conner Humphreys",
        "id": 6
      },
      {
        "name": "Mykelti Williamson",
        "id": 4
      },
      {
        "name": "Robin Wright",
        "id": 2
      },
      {
        "name": "Sally Field",
        "id": 5
      },
      {
        "name": "Tom Hanks",
        "id": 1
      }
    ],
    "length": 8520,
    "id": 1,
    "director": "Robert Zemeckis",
```

```
    "title": "Forrest Gump"
  }
]
```

Obtener el recurso Película con id 1

Realiza una petición `GET` a la URL `http://localhost:5000/api/v1.0/films/1`. La respuesta será:

```
{
  "year": 1994,
  "actors": [
    {
      "name": "Tom Hanks",
      "id": 1
    },
    {
      "name": "Robin Wright",
      "id": 2
    },
    {
      "name": "Gary Sinise",
      "id": 3
    },
    {
      "name": "Mykelti Williamson",
      "id": 4
    },
    {
      "name": "Sally Field",
      "id": 5
    },
    {
      "name": "Michael Conner Humphreys",
      "id": 6
    }
  ],
  "length": 8520,
  "id": 1,
  "director": "Robert Zemeckis",
  "title": "Forrest Gump"
}
```

Este tutorial ha sido una guía de introducción a cómo implementar un API REST en Python con Flask. No obstante, todavía hay muchos puntos de mejora y cosas por implementar. Por ejemplo:

- Gestión de usuarios.

- Securizar el acceso a los endpoints con tokens JWT.
- Generar la documentación del API.
- Habilitar CORS si el API se va a consumir desde un dominio diferente al de su despliegue.
- Validar correctamente el JSON de entrada.
- Endpoints para modificar y/o eliminar recursos.
- Mejora de la gestión de errores.