WebApi Polls

Documentación

By Dan Milgram

[Enunciado 2](#_Toc45377777)

[Stack de tecnología utilizado 2](#_Toc45377778)

[Documentación de los EndPoints 3](#_Toc45377779)

[Deployando la API con Docker 5](#_Toc45377780)

[Posibles mejoras 6](#_Toc45377781)

[¿Qué fue lo más desafiante? 6](#_Toc45377782)

# Enunciado

Según el enunciado se planteó la realización de un sistema de encuestas. Luego de analizarlo, se logró determinar los siguientes casos de uso ordenados por prioridad:

* Alta de encuesta [poll] 🡪 se carga una encuesta con los campos: name, expiration, questions(junto con answers) y tags
* Responder encuesta 🡪 cualquier persona puede responder una encuesta indicando su mail y eligiendo una opción válida por cada pregunta.
* SignUp, Login, Logout 🡪 módulo login
* Tags 🡪 Agregado de etiquetas para luego poder clasificar las encuestas
* Listado de encuestas 🡪 Devuelve todas las encuestas cargadas hasta el momento
* Historial por usuario 🡪 Devuelve las encuestas que realizó una persona en particular

# Stack de tecnología utilizado

Se utilizó el framework Flask que permite crear una API de forma ágil, sumada a la base MongoDB para la persistencia de los datos.

La arquitectura de la solución consta de los siguientes componentes que interactúan entre sí:

* App.py
  + Inicializa la app junto con el loginmanager, mongoDB y Swagger, y registra un blueprint para poder modularizar las diferentes routes.
* Routes.
  + Define los Endpoints de la aplicación, divididos en un archivo .py por cada conjunto de rutas para cada entidad [polls, answers, login, tags]. Recibe los requests HTTP del usuario y hace de controlador realizando la lógica de negocio necesaria. Finalmente llama al dbService para efectuar las operaciones CRUD.
* Services
  + Permite el acceso a los datos, ofreciendo la clase MongoAPI con un conjunto de métodos para realizar operaciones sobre la DB.
* Validators.
  + Archivos con módulos que dan apoyo a routes, realizando las validaciones pertinentes para que la información persisitida en la base de datos sea exactamente lo que se requiere guardar.

# Documentación de los EndPoints :

**Historia de usuario relacionada: alta de usuario**

Endpoint: /signup

Metodo HTTP: POST

Ejemplo de JSON schema válido:

{

    "name":"Dan Milgram",

    "email": "ddmilgram@gmail.com",

    "password": "dancito"

}

Endpoint: /login

Metodo HTTP: POST

Historia de usuario relacionada: alta de usuario

Ejemplo de JSON schema válido:

{

"email": "ddmilgram@gmail.com",

"password": "dancito"

}

Endpoint: /logout

Metodo HTTP: POST

Historia de usuario relacionada: alta de usuario

Ejemplo de JSON schema válido: -

**Historia de usuario relacionada: alta de encuesta**

Endpoint: /polls/add

Metodo HTTP: POST

Login requerido: Si

Ejemplo de JSON schema válido:

{

    "name": "Encuesta ",

    "expiration": "10/12/2020",

    "questions": {"¿Te gustan las encuestas?": ["Si", "No"], "¿Cuantas encuestas haces por año?" : ["1","2"]},

    "tags":["soyetiqueta"]

}

**Historia de usuario relacionada: responder encuesta**

Endpoint: /answers/add

Metodo HTTP: POST

Login requerido: No

Ejemplo de JSON schema válido:

{

    "pollid": "5f08e7085e1b7b126ff88234",

    "name": "Encuesta loca",

    "answers": {"¿Te gustan las encuestas?":"Si", "¿Cuantas encuestas haces por año?":"2" },

    "email": "ddmilgram@gmail.com"

}

**Historia de usuario relacionada: Historial por usuario**

Endpoint: /answers/answersbyuser

Metodo HTTP: POST

Login requerido: Si

Ejemplo de JSON schema válido:

{

    "email":"ddmilgram@gmail.com"

}

**Historia de usuario relacionada: Listado de encuestas**

Endpoint: /polls

Metodo HTTP: GET

Login requerido: Si

Ejemplo de JSON schema válido: -

**Historia de usuario relacionada: Agregar etiqueta**

Endpoint: /tags/add

Metodo HTTP: POST

Login requerido: Si

Ejemplo de JSON schema válido:

{

    "tag":"etiqueta grosa"

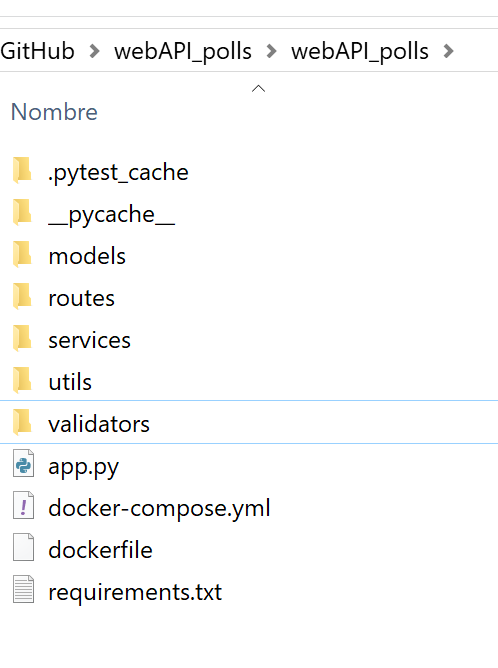
}

Colección postman : <https://www.getpostman.com/collections/3871d22fb2936ee081e2>

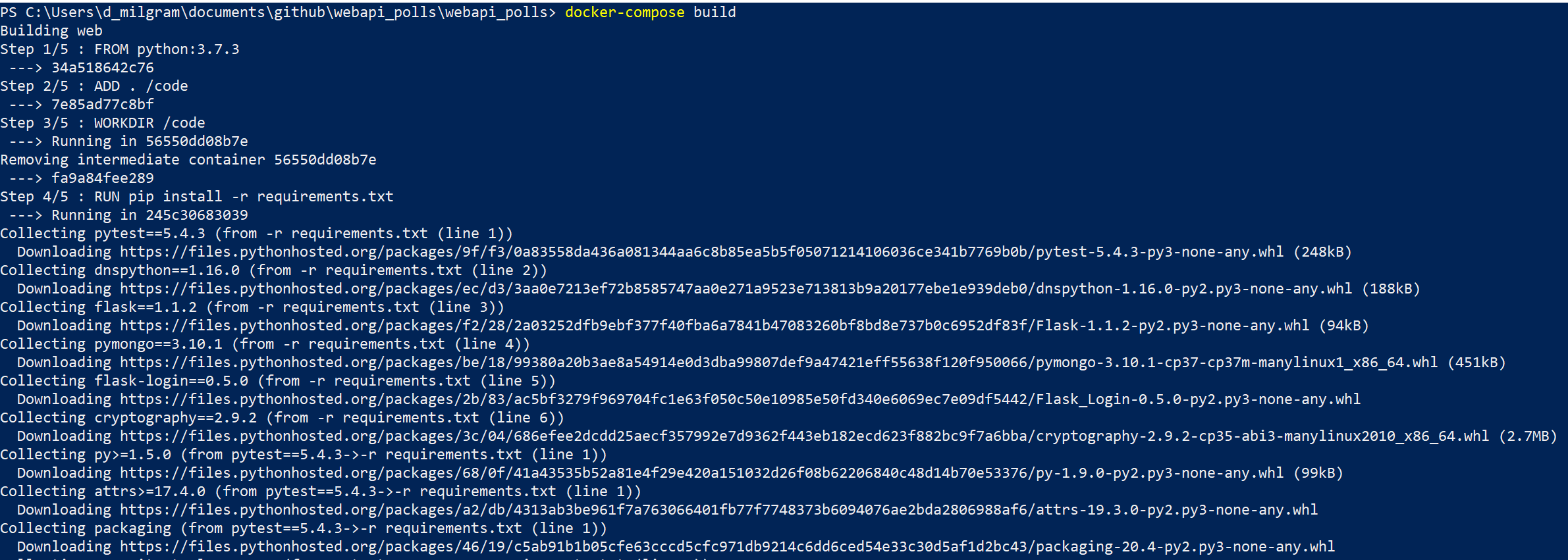
# Deployando la API con Docker

Para levantar la API utilizando un contenedor Docker, se debe realizar los siguientes pasos:

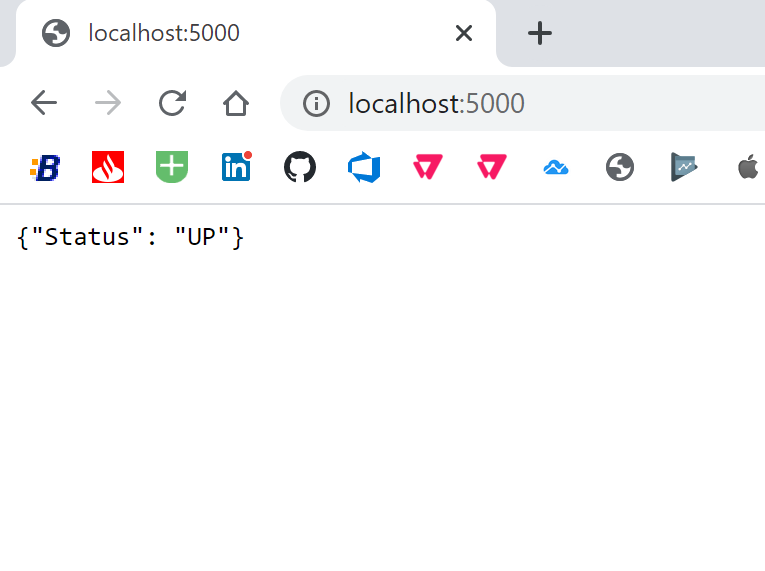
1. Desde la consola de comandos, situarse sobre la carpeta WebAPI\_Polls



1. Ejecutar Docker-compose build 🡪 Esto va a tomar el archivo dockerfile y va a generar la imagen de la solución, instalando antes todos los módulos requeridos en el archivo requirements.txt



1. Ejecutar Docker run -p 5000:5000 webapi\_polls\_web 🡪 Inicia el contenedor en el puerto 5000.
2. Verificar que la API esté arriba

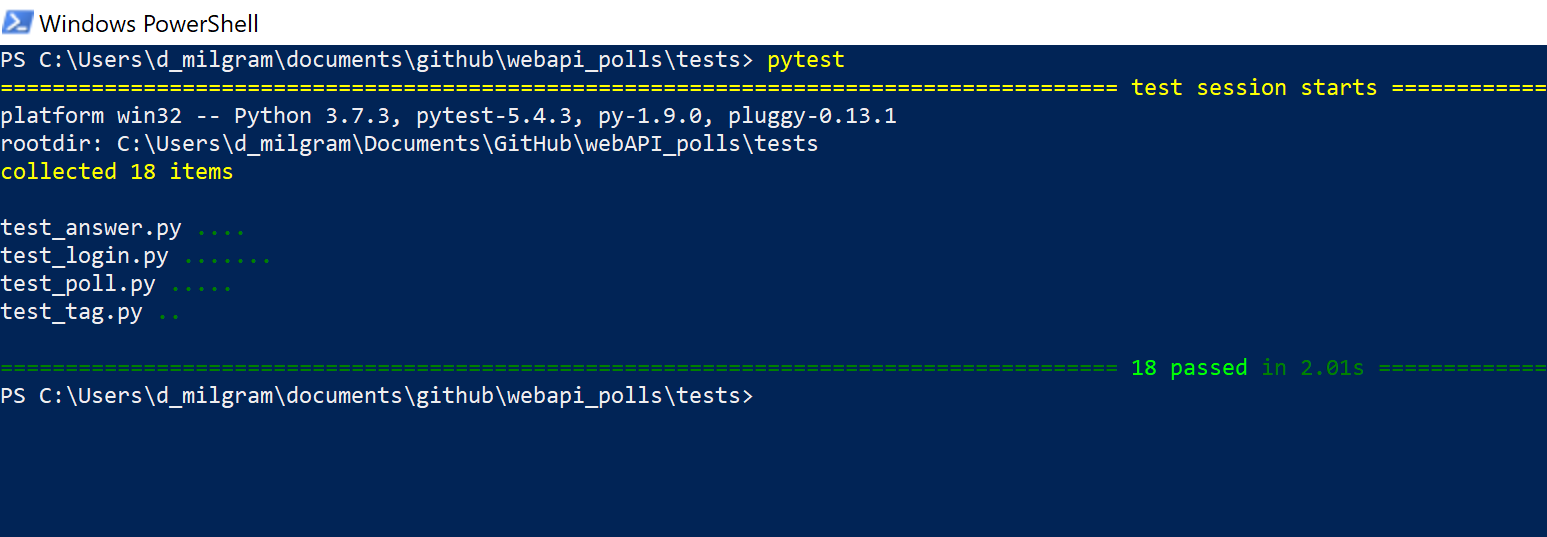


# Tests unitarios

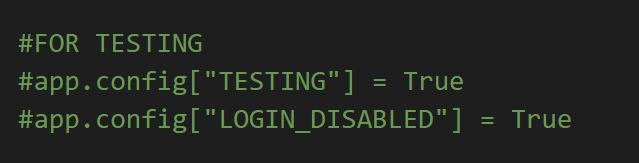
Para testear los endpoints con la librería PyTest, se crearon todos los tests dentro del directorio “tests”.

Para ejecutarlos:

1. Asegurarse que esté la API deployada y escuchando en localhost:5000
2. Pararse sobre la carpeta tests y ejecutar pytest



Para que los test se ejecuten correctamente, se debe tener descomentadas las siguientes líneas del archivo app.py



De lo contrario los test van a fallar ya que se va a solicitar autenticación..

# Posibles mejoras

* Seguridad : Tener en cuenta que la versión de la API lograda está preparada para un entorno de Desarrollo, y no aún para producción, ya que no está segurizada la URL del connectionString. A fines de un ejercicio de pruebas esto no es un inconveniente, pero si el plan fuese montar la API en producción, necesitaríamos resolver este punto.
* Uso de Swagger: para dar más claridad a los endPoints de la API, se podría hacer uso de Swagger como URL de documentación de los EndPoints. Aquí un ejemplo de su utilización sobre Flask <https://medium.com/@sean_bradley/add-swagger-ui-to-your-python-flask-api-683bfbb32b36>

# ¿Qué fue lo más desafiante?

Encontré dos puntos interesantes

Por un lado fue un lindo desafío montar el módulo de login utilizando flask-login, ya que toda la documentación de esta librería que está en internet apunta a una web FrontEnd que llama al login desde un formulario, mientras que en este caso precisaba utilizarlo de otra forma. Hubo entonces que leer con detalle la documentación de la librería para poder adaptarla a la necesidad concreta de este ejercicio.

Por otro lado, al momento de resolver la validación de las respuetas a una encuesta, había que validar que para cada pregunta respondida, cada respuesta ofrecida por el usuario esté contenida dentro de las posibles respuestas.

Para eso se creo el método **ValidateQuestionsAndAnswers** dónde de una forma muy “Pythonic way”, en una línea de código verificamos que cada pregunta-respuesta sea válida comparada con la encuesta que está contestando el usuario. Esto fue muy interesante.

def ValidateQuestionsAndAnswers(pollQuestions, answeredQuestions):

    try:

        shared\_items = {k: pollQuestions[k] for k in pollQuestions if k in answeredQuestions and answeredQuestions[k] in pollQuestions[k]}

        if len(shared\_items) == len(pollQuestions) ==len(answeredQuestions):

            return ok()

        else:

            return notValidAnswers()

    except Exception as e:

        return notValidAnswers()