

ETL Project II: Docker

Aplicación web con Flask y MongoDB



Profesor: Jorge Peralta Dolz

Escuela: La Salle Bonanova

Máster: Data Science

Asignatura: Infraestructuras de computación



Índice

Introducción.....	3
Arquitectura	3
Dockerfile	4
Docker-Compose.....	5
Desarrollo – Flujo	7
Dificultades encontradas	18
Bibliografía.....	19

Introducción

En este proyecto se explicará el funcionamiento de una aplicación llamada, “Supergestor Marvel”, en Flask conectada con el sistema gestor de base de datos MongoDB, sobre el lenguaje de programación Python, además esta aplicación se automatizará mediante Docker Compose aplicando conceptos de microservicios.

El esqueleto de este proyecto ha sido obtenido a través de la plataforma Crashell, donde siguiendo este pequeño tutorial se ha ido entendiendo, paso a paso, el funcionamiento de la aplicación inicial.

Una vez entendido el proyecto base, se han realizado una serie de cambios que han permitido crear “Supergestor Marvel”:

“Supergestor Marvel” es una aplicación web que permite al usuario buscar cualquier super héroe de Marvel que se encuentre dentro de la Web Api de Marvel, y después en base al número de comics, series, historias y eventos el usuario decide si desea guardarlo en su base de datos o no.

Arquitectura

En la *Ilustración 1* se puede observar la arquitectura creada mediante Docker-Compose. Ambos contenedores se encuentran en la misma red “*default*”. El contenedor web internamente usa el puerto 5000 y se accede a él a través del puerto 8000, es decir, el tráfico que llega al puerto 8000 de la máquina host se está redirigiendo al puerto 5000 del contenedor Docker. Por lo tanto, si desde la máquina virtual se quiere acceder a la web se tendrá que usar la notación “localhost:8000/”.

Por otro lado, la base de datos abre los puertos en el rango 2701[7-9] con el rango homónimo 2701[7-9] que se encuentra en el contenedor, por lo tanto, serán expuestos los mismos puertos. Además, se crean diferentes volúmenes para persistir datos y además con el archivo “mongo-init.js” se crea un usuario con sus respectivas credenciales además de una base de datos.

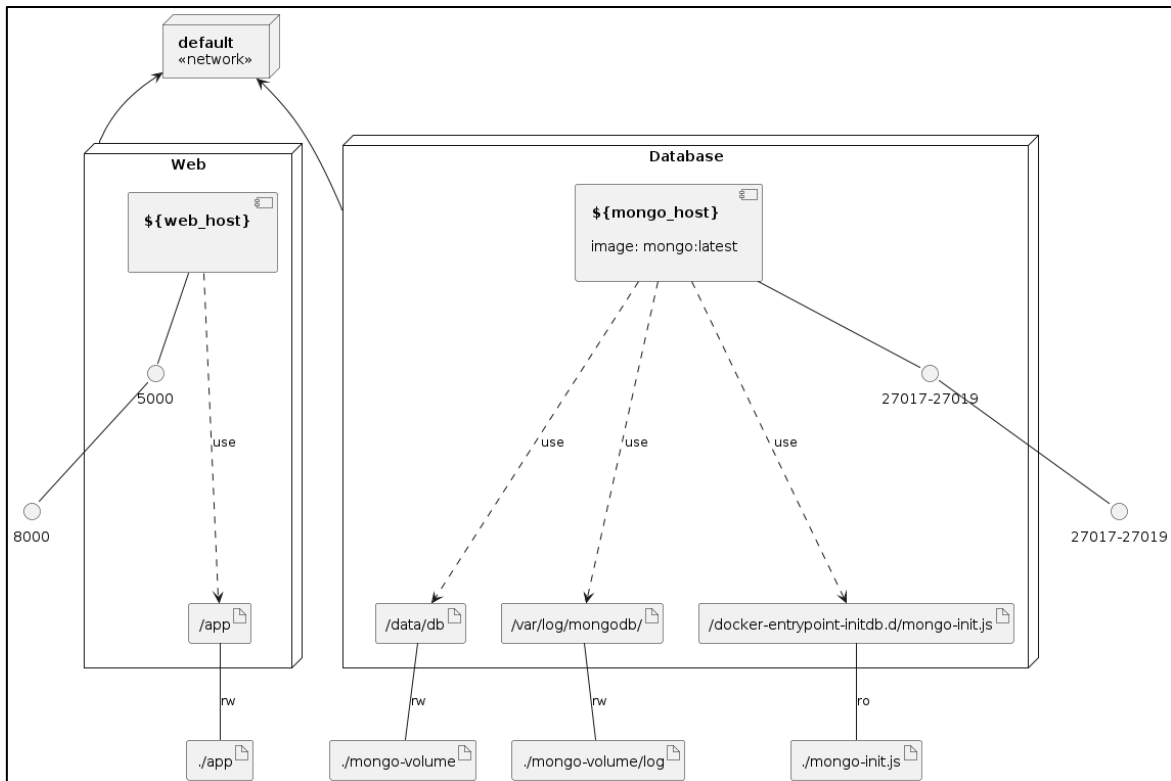


Ilustración 1

Dockerfile

El Dockerfile original es el que se muestra en la *Ilustración 2*, se basa en `Python:3.7-alpine` que es una distribución Linux ligera que permite el funcionamiento del aplicativo. Se probó de cambiar a la última versión estable reducida, `Python:3.12-slim`, pero esta no funcionaba.

En la línea 5 se crea dentro del contenedor un directorio `/app` y acto seguido se declara este como directorio de trabajo. A continuación, en la línea 7 se copia todo el directorio desde donde se está ejecutando el Dockerfile al directorio destino `/app`.

El paso de la línea numero 9 ha sido eliminado para la “Supergestor Marvel”, ya que se trata de la instalación del compilador gcc y de las cabeceras de Linux que para el proyecto no son necesarias:

- No hay dependencias de Python con extensiones C.
- No se usan extensiones de Flask que requieran compilación.
- No se compilan recursos Frontend.
- En el caso de las cabeceras de Linux, no se interactúa directamente con el kernel.

Por último, se instalan los requerimientos necesarios con la línea 10 (*Ilustración 3*) y se expone el contendor para que escuche en el puerto 5000. Como requerimientos necesarios básicos, se tiene *Flask* ya que esté si que se necesita para el desarrollo de la aplicación, *requests* que permite hacer las llamadas a la api de Marvel y *pymongo* que permite comunicar MongoDB con Python. La librería *xmltodict* no se ha usado, pero se ha considerado dejarla ya que a futuro podría ser de gran utilidad manejar data XML de forma mucho más sencilla.

```
1 # syntax=docker/dockerfile:1
2
3 FROM python:3.7-alpine
4
5 RUN mkdir /app
6 WORKDIR /app
7 COPY . /app
8
9 RUN apk add --no-cache gcc musl-dev linux-headers
10 RUN pip install -r requirements.txt
11
12 EXPOSE 5000
```

Ilustración 2

```
ETL > app > requirements.txt
1 Flask==2.1.2
2 requests==2.25.1
3 xmltodict==0.13.0
4 pymongo==4.1.1
```

Ilustración 3

Docker-Compose

Una vez entendido el Dockerfile que construirá la imagen sobre la cual se construirá el contendor encargado de ejecutar la aplicación Flask, se mostrará como a través de Docker-Compose se construyen y se conectan estos dos contenedores. Para ello, vienen declaradas unas variables de entorno predefinidas en un archivo llamado *.env* (*Ilustración 5*).

```
ETL > docker-compose.yml
1  version: '3.9'
2  services:
3    web:
4      env_file:
5        - .env
6      container_name: ${WEB_HOST}
7      hostname: ${WEB_HOST}
8      build: ./app
9      entrypoint:
10       - flask
11       - run
12       - --host=0.0.0.0
13     environment:
14       FLASK_DEBUG: 1
15       FLASK_APP: ./app.py
16       FLASK_RUN_HOST: 0.0.0.0
17       TEMPLATES_AUTO_RELOAD: 'True'
18       FLASK_ENV: development
19     ports:
20       - '8000:5000'
21     links:
22       - database
23     depends_on:
24       - database
25     volumes:
26       - ./app:/app
27     networks:
28       - default
```

Ilustración 4

```
ETL > .env
1  WEB_HOST=cs_api
2
3  MONGO_HOST=cs_mongodb
4  MONGO_PORT=27017
5  MONGO_USER=root-crashell
6  MONGO_PASS=password-crashell
7  MONGO_DB=db_crashell
```

Ilustración 5

En la *Ilustración 4* aparece la construcción del contenedor web donde se usan las variables de entorno para declarar los nombres, además en vez de usar la *images* y declarar una imagen, se indica que la imagen que se usara es la que se tiene que construir y que se encuentra en el directorio *./app* es decir, se construirá en base al Dockerfile que se ha comentado anteriormente. Se especifican ciertos parámetros para el entorno de Flask que permiten el desarrollo más eficiente y cómodo, se declara la conexión de puertos antes mencionada en la sección y se especifica que esta vinculada al otro servicio que se verá a continuación que es nuestra base de datos MongoDB (*database*). Por último, se especifica que depende de la base de datos, se crea un volumen para persistir los datos y que estos estén sincronizados y se especifica la red en la que se encontrará este servicio (*default*) , la misma que la base de datos.

Por otro lado, en la *Ilustración 5* aparece la construcción del contenedor de la base de datos. En este caso la imagen si que es la de *mongo:latest* para poder obtener la última versión estable. De igual forma que en el contenedor web, también se declaran los nombres y ciertas variables a través de las variables de entorno, también se crean volúmenes para persistir datos y se declaran los puertos como se ha comentado al principio de la sección y se establece la red *default*.

```
29   database:
30     image: mongo:latest
31     env_file:
32       - .env
33     container_name: ${MONGO_HOST}
34     hostname: ${MONGO_HOST}
35     environment:
36       MONGO_INITDB_ROOT_USERNAME: ${MONGO_USER}
37       MONGO_INITDB_ROOT_PASSWORD: ${MONGO_PASS}
38       MONGO_INITDB_DATABASE: ${MONGO_DB}
39     volumes:
40       - ./mongo-init.js:/docker-entrypoint-initdb.d/mongo-init.js:ro
41       - ./mongo-volume:/data/db
42       - ./mongo-volume/log:/var/log/mongodb/
43     ports:
44       - '27017-27019:27017-27019'
45     networks:
46       - default
47   volumes:
48     persistent:
```

Ilustración 4

Desarrollo – Flujo

Con tal de hacer una explicación de principio a fin, se explicará paso a paso todo el flujo de funcionamiento de la aplicación y paralelamente se mencionarán por encima fragmentos de código para complementar la explicación.

En primer lugar, suponiendo que el repositorio de la aplicación “Supergestor Marvel” esté disponible para el usuario, éste tendrá que clonar el repositorio. Una vez clonado, deberá de lanzar el comando “docker-compose up -d” para que estos contenedores se levanten y se ejecuten en segundo plano (en la *Ilustración 7* se ejecuta el comando con el Dockerfile original, y en la *Ilustración 8* se ejecuta el comando con la versión final personalizada, eliminando la instalación necesaria del compilador y de las cabeceras Linux, entre otros, y se observa como disminuye considerablemente el tamaño.

```
=> => writing image sha256:fac5269dfc686f94837238c0c71d1da9319f6b895b6d09f38437bebd388020f1
=> => naming to docker.io/library/etl-web
[+] Running 2/2
✓ Container cs_mongodb Started
✓ Container cs_api Started
root@ETL:/home/didac/Documents/MyLocalRepo/Data-Science-Master/ETL# docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
etl-web latest fac5269dfc68 41 minutes ago 222MB
mongo latest 2e123a0ccb4b 11 days ago 757MB
root@ETL:/home/didac/Documents/MyLocalRepo/Data-Science-Master/ETL#
```

Ilustración 7

```
=> => writing image sha256:382118b550acbf3271407ce52628442b2fdd497e7b77635113d12e54092e6753
=> => naming to docker.io/library/etl-web
[+] Running 2/2
✓ Container my_mongo Started
✓ Container my_api Started
root@ETL:/home/didac/Documents/MyLocalRepo/Data-Science-Master/ETL# docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
etl-web latest 382118b550ac 10 minutes ago 68.3MB
mongo latest 2e123a0ccb4b 11 days ago 757MB
root@ETL:/home/didac/Documents/MyLocalRepo/Data-Science-Master/ETL#
```

Ilustración 8

Una vez los contenedores están en marcha, ya se puede entrar en el navegador para acceder a la web “localhost:8000/”.

La *Ilustración 9* muestra la vista de la página web principal con el inspector abierto, la *Ilustración 10* muestra el *body* de esta vista y la 11 como ésta interactúa con Flask y devuelve el templete renderizado.

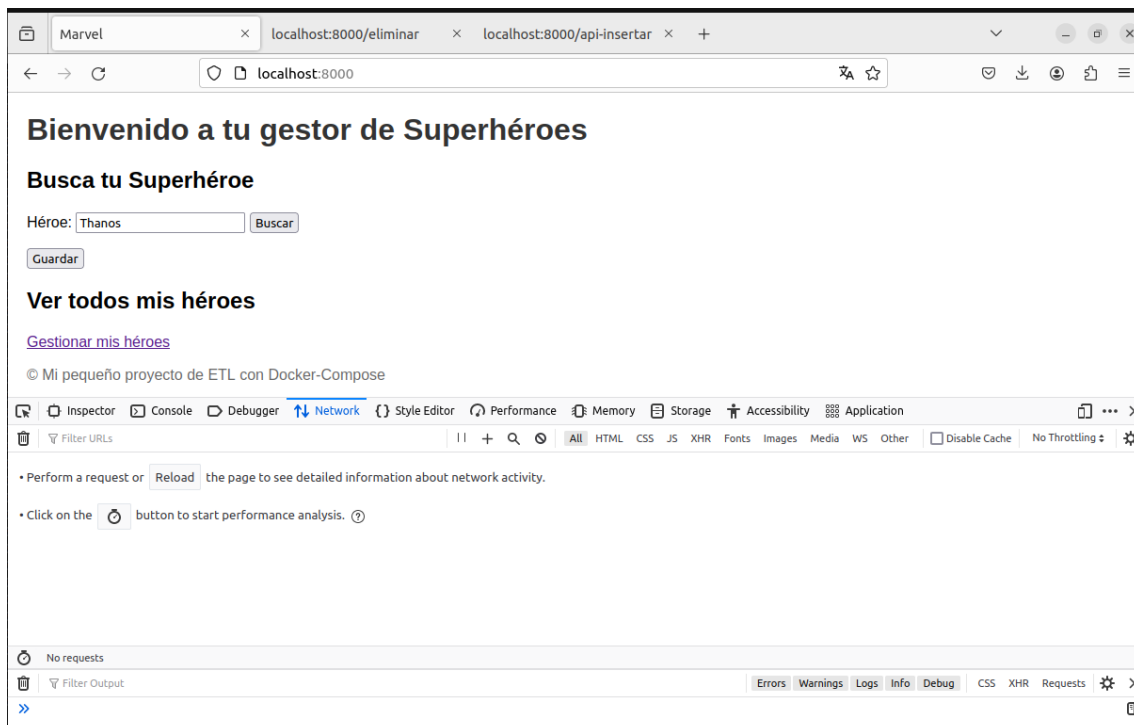


Ilustración 9


```

87 <body>
88   <header>
89     <h1>Bienvenido a tu gestor de Superhéroes</h1>
90   </header>
91   <section id = "FormularioBusqueda">
92     <h2>Busca tu Superhéroe</h2>
93     <form id="buscarhero">
94       <label for="hero">Héroe:</label>
95       <input type="text" name="hero" id="hero">
96       <button type="button" onclick="sendHero()"> Buscar </button>
97     </form>
98   </section>
99
100   <section id = "RespuestasApiMongo">
101     <p id="respuestaApi"></p>
102     <p id="respuestaMongoDB"></p>
103     <button type="button" onclick="saveHero()"> Guardar </button>
104     <div id="temporal"></div>
105   </section>
106
107   <section id = "listarSuperheros">
108     <h2>Ver todos mis héroes</h2>
109     <a href="{{url for('list_heros')}}"> Gestionar mis héroes</a>
110   </section>
111
112   <footer>
113     <p>&copy; Mi pequeño proyecto de ETL con Docker-Compose</p>
114   </footer>
115 </body>

```

Ilustración 10

```

109
110 @app.route('/', methods=('GET', 'POST'))
111 def index():
112     return render_template('index.html')
113

```

Ilustración 11

A parte de la *url* principal, también se disponen de dos *urls* extras que han servido de ayuda durante el desarrollo.

La primera es “localhost:8000/eliminar”, esta básicamente cada vez que la llamas elimina todos los elementos de la colección de la base de datos que se está utilizando en la aplicación (*Ilustración 12 para vista y 13 para código*).

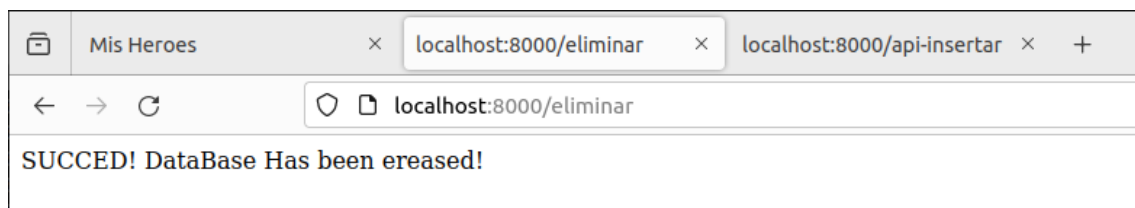


Ilustración 12

```
100 @app.route('/eliminar')
101 def delete_all():
102     ok, connection = getMongoClient()
103     if not ok:
104         return Response(response=json.dumps(connection), mimetype='application/json')
105     collection = getCollection(connection,"superheros")
106     collection.delete_many({})
107     return "SUCCED! DataBase Has been ereased!"
```

Ilustración 13

En la *Ilustración 13* se aparecen dos funciones globales, que viene definidas por la *Ilustración 14*.

```
73
74 def getMongoClient():
75     return ConnectionMongoDB(DB_SERVER,DB_PORT,DB_USER,DB_PASS,DB_NAME).getDB()
76
77 def getCollection(connection,collection_name):
78     db = connection[DB_NAME]
79     return db[collection_name]
80
```

Ilustración 14

Así pues, desde nuestra pagina principal (*Ilustración 9*) haciendo clic en el link hacía “Gestionar mis Héroes”, éste conduce hacia una página (“localhost:8000/list_heros”) donde se consultan y listan todos los superhéroes guardados en la colección de la base de datos (*Ilustración 15 para vista y 16 para código*).



Mis Heroes x localhost:8000/eliminar x localhost:8000/api-insertar x +

localhost:8000/list_heros

Lista de Superhéroes

Nombre	Descripción	Número de Comics	Número de Series	Número de Historias	Número de Eventos
--------	-------------	------------------	------------------	---------------------	-------------------

Buscar nuevo Superhéroe

[Ir a la pagina principal](#)

© Mi pequeño proyecto de ETL con Docker-Compose

Ilustración 15

```
113
114 @app.route('/list_heros', methods=('GET', 'POST'))
115 def list_heros():
116     ok, connection = getMongoClient()
117     if not ok:
118         return Response(response=json.dumps(connection), mimetype='application/json')
119     collection = getCollection(connection,"superheros")
120     cursor_list = collection.find()
121     return render_template('heros.html', allHeros=cursor_list)
```

Ilustración 16

La segunda *url* que ha ayudado mucho para el desarrollo del proyecto ha sido “localhost:8000/api-insert”. Esta *url* inserta unos superhéroes inventados por defecto dentro del código y después lista de forma encadenada y sin espacios los nombres de todos los superhéroes de la colección de la base de datos. De esta manera, permitió testear para el correcto guardado en la base de datos y para el correcto leído (*Ilustración 17 para los superhéroes inventados, 18 para la vista y 19 para el código*).

```
63
64 #Custom Hereos for testing
65 MisSuperheros = [
66     Superhero("Juan", "mortal",23, 3, 6, 6),
67     Superhero("Julio", "lento",254, 2, 41, 2),
68     Superhero("Cesar", "rapido",2355, 3, 49, 1),
69     Superhero("Patroclo", "inutil",234, 9, 43, 8)
70 ]
```

Ilustración 17

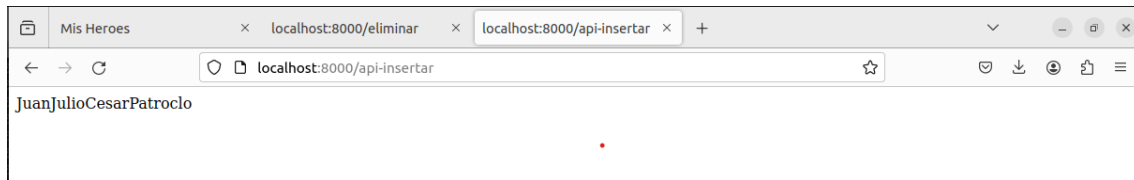



Ilustración 18

```
82
83 @app.route('/api-insertar')
84 def get_api():
85     ok, connection = getMongoClient()
86     if not ok:
87         return Response(response=json.dumps(connection), mimetype='application/json')
88     collection = getCollection(connection,"superheros")
89
90     for sh in MisSuperheros:
91         collection.insert_one(sh.toDBCollection())
92
93     cursor_list = collection.find()
94     myList = []
95     [myList.append(i['nombre']) for i in cursor_list]
96
97     finalStr = "".join(myList)
98     return finalStr
```

Ilustración 19

Una vez guardados estos superhéroes en la base de datos, volvemos a “localhost:8000/list_heros” y vemos como ahora si que estos ya se muestran (*Ilustración 20*).



The screenshot shows a web browser with three tabs: 'Mis Heroes', 'localhost:8000/eliminar', and 'localhost:8000/api-insertar'. The address bar shows 'localhost:8000/list_heros'. The page title is 'Lista de Superhéroes'. Below the title is a table with the following data:

Nombre	Descripción	Número de Comics	Número de Series	Número de Historias	Número de Eventos
Juan	mortal	23	3	6	6
Julio	lento	254	2	41	2
Cesar	rapido	2355	3	49	1
Patroclo	inutil	234	9	43	8

Below the table is a section titled 'Buscar nuevo Superhéroe' with a link 'Ir a la pagina principal'. At the bottom, it says '© Mi pequeño proyecto de ETL con Docker-Compose'.

Ilustración 20

Si ahora, volvemos por un momento a la *Ilustración 9*, donde se ve que en el texto esta escrito el superhéroe “Thanos” y le damos al botón de buscar, la función de JavaScript de la *Ilustración 21* recolectara la información la enviará a nuestra aplicación Flask y ésta realizará una llamada a la api de Marvel devolviendo en caso de que el superhéroe exista, el nombre, el número de comics, de series, de historias y de eventos (*Ilustración 22*). Una vez esta información sea devuelta al Frontend, se rellenará la web con el contenido (*Ilustración 23*).

```
function sendHero(){
  //Obtener el valor de la palabra
  var word = document.getElementById("hero").value;
  fetch('/process_hero',{
    method: 'POST',
    headers:{
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({hero: word})
  })
  .then(response => response.json())
  .then(data=> {
    console.log('Respuesta del servidor:', data.resultado);

    //mostrar respuesta en la web
    var serverResponse = document.getElementById("respuestaApi");
    document.getElementById('temporal').innerHTML = JSON.stringify(data,null,2);
    document.getElementById('respuestaMongoDB').innerText = "";
    document.getElementById('hero').value = "";
    serverResponse.innerText = "El/La Superhéroe " +
    Sdata.nombre + " tiene:\n" +
    "Comics: " + data.comics + "\n" +
    "Series: " + data.series + "\n" +
    "Historias: " + data.historias + "\n" +
    "Eventos: " + data.eventos + "\n"
  })
  .catch(error => {
    console.error('Error al enviar el titulo al servidor:', error);
  });
};
```

Ilustración 21

```
122
123 @app.route('/process_hero', methods=['POST'])
124 def buscar_super_hero():
125     miSuperhero = request.json.get('hero')
126
127     url = 'https://gateway.marvel.com/v1/public/characters?'
128     public_key = '9868ba90ad5b9997dde549a64cf6ada7'
129     private_key = 'dde581eba9a6d29eafdcfb2330bda2b13c0ccb4c'
130     ts = str(time.time()).split('.')[0]
131     md5_hash = hashlib.md5((ts+private_key+public_key).encode()).hexdigest()
132
133     url = url + 'name=' + miSuperhero + '&ts=' + ts + '&apikey=' + public_key + '&hash=' + md5_hash
134
135     response = requests.get(url)
136     if response.status_code == 200:
137
138         mihero = response.json()[0]['data']['results'][0]
139
140         nombre = mihero['name']
141         descripcion = mihero['description'] or "No description!"
142         comics = mihero['comics']['available']
143         series = mihero['series']['available']
144         historias = mihero['stories']['available']
145         eventos = mihero['events']['available']
146
147         miheroClass = Superhero(nombre,descripcion,comics,series,historias,eventos)
148
149     else:
150         print(f"Error: {response.status_code}")
151
152     return jsonify({
153         'nombre': miheroClass.nombre,
154         'descripcion': miheroClass.descripcion,
155         'comics': miheroClass.comics,
156         'series': miheroClass.series,
157         'historias': miheroClass.historias,
158         'eventos': miheroClass.eventos
159     })
```

Ilustración 22

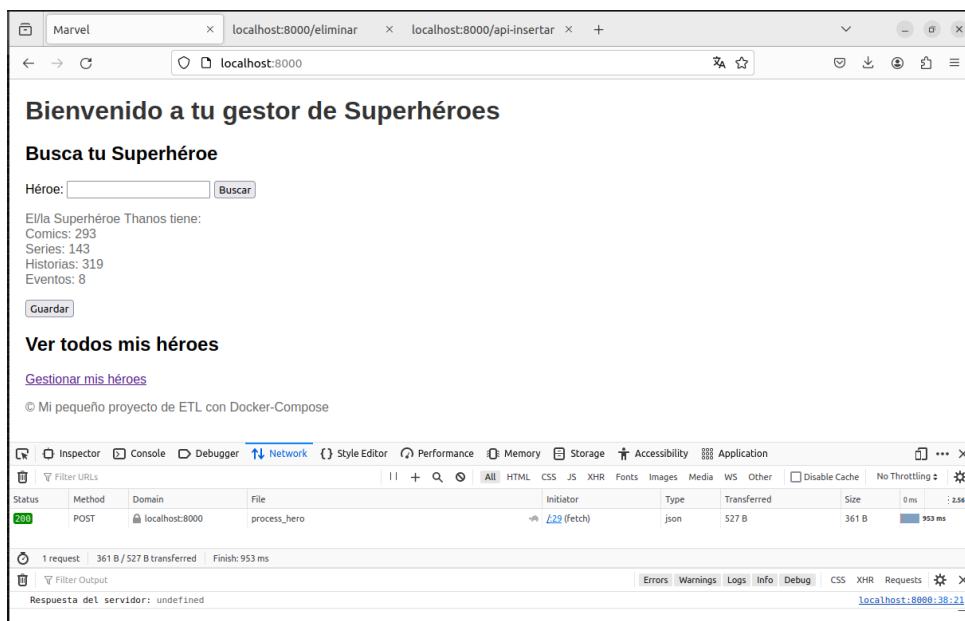


Ilustración 23

Ahora, el usuario es el encargado de escoger si desea guardar este superhéroe en su base de datos o no. En caso afirmativo, deberá pulsar el botón de “Guardar” que activará la función de JavaScript de la *Ilustración 24* que se conectará con la parte del Backend de Flask tal y como indica la *Ilustración 25*. El resultado en la vista quedará tal y como indica la *Ilustración 26*.

```
function saveHero(){
  var hero = document.getElementById("temporal").innerText;
  if (!hero) {
    alert('No hay super hero para guardar!')
  }else {
    fetch('/save_hero',{
      method: 'POST',
      headers:{
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(hero)
    })
    .then (response => response.text())
    .then (data => {
      document.getElementById('respuestaMongoDB').innerText = data;
      document.getElementById('respuestaApi').innerText = "";
      document.getElementById('temporal').innerText = "";
    })
    .catch (error => {
      console.error('Error al guardar en MongoDB:', error);
    });
  };
};
```

Ilustración 24

```
160
161 @app.route('/save_hero', methods=['POST'])
162 def guardar_hero():
163     ok, connection = getMongoClient()
164     if not ok:
165         return Response(response=json.dumps(connection), mimetype='application/json')
166     collection = getCollection(connection,"superheros")
167
168     mihero = Superhero(**json.loads(request.json))
169     collection.insert_one(mihero.toDBCollection())
170     return "Superhéroe Guardado!"
171
```

Ilustración 25

The screenshot shows a web browser window with the URL `localhost:8000`. The page title is "Bienvenido a tu gestor de Superhéroes". It features a search bar labeled "Busca tu Superhéroe" with a "Buscar" button. Below the search bar, it says "Superhéroe Guardado!" and has a "Guardar" button. There is a link "Ver todos mis héroes" and a footer "© Mi pequeño proyecto de ETL con Docker-Compose".

The bottom part of the image shows the Chrome DevTools Network tab. It displays two requests:

Status	Method	Domain	File	Initiator	Type	Transferred	Size	ms
200	POST	localhost:8000	process_hero	L23 (Fetch)	json	527 B	361 B	953 ms
200	POST	localhost:8000	save_hero	L63 (Fetch)	html	194 B	21 B	

The summary at the bottom indicates "2 requests | 382 B / 721 B transferred | Finish: 1.53 min". The console shows "Respuesta del servidor: undefined".

Ilustración 26

Cuando se ha guardado el superhéroe el usuario puede ir a su página de gestión de superhéroes y puede ver que efectivamente Thanos, se ha guardado correctamente (*Ilustración 27*).

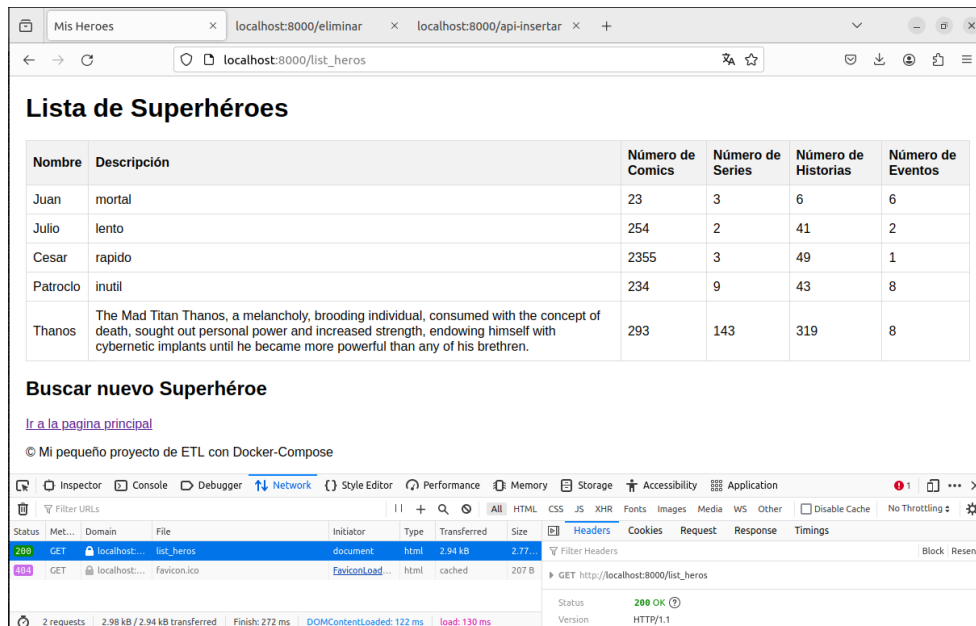


Ilustración 27

El *body* de la vista que lista los superhéroes queda definido en la *Ilustración 28*.

```
<body>
<h1>Lista de Superhéroes</h1>
<table>
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Descripción</th>
      <th>Número de Comics</th>
      <th>Número de Series</th>
      <th>Número de Historias</th>
      <th>Número de Eventos</th>
    </tr>
  </thead>
  <tbody>
    {% for hero in allHeros %}
      <tr>
        <td>{{ hero['nombre'] }}</td>
        <td>{{ hero['descripcion'] }}</td>
        <td>{{ hero['comics'] }}</td>
        <td>{{ hero['series'] }}</td>
        <td>{{ hero['historias'] }}</td>
        <td>{{ hero['eventos'] }}</td>
      </tr>
    {% endfor %}
  </tbody>
</table>
<section>
  <h2>Buscar nuevo Superhéroe</h2>
  <a href="{{url_for('index')}}"> Ir a la pagina principal</a>
</section>
<footer>
  <p>&copy; Mi pequeño proyecto de ETL con Docker-Compose</p>
</footer>
</body>
```

Ilustración 28

Para que todo el desarrollo sea posible y eficiente se han trabajado con dos clases:

- Clase *Superhero*: permite guardar la información necesaria del superhéroe y manejarla de forma rápida y sencilla para su guardado y listado (*Ilustración 29*).
- Clase *ConnectionMongoDB*: permite realizar la conexión con la base de datos o devolver una excepción en caso de error (*Ilustración 30*)

```
37 @dataclass
38 class Superhero:
39     nombre: str
40     descripcion: str
41     comics: int
42     series: int
43     historias: int
44     eventos: int
45
46     def toDBCollection (self):
47         return {
48             "nombre":self.nombre,
49             "descripcion": self.descripcion,
50             "comics":self.comics,
51             "series": self.series,
52             "historias": self.historias,
53             "eventos": self.eventos
54         }
```

Ilustración 29

```
10 #CLASSES
11 @dataclass
12 class ConnectionMongoDB:
13     server: str
14     port: str
15     username: str
16     password: str
17     db: str
18
19     def getDB(self):
20         mongoClient = MongoClient("mongodb://" +
21                                   str(self.username) + ":" +
22                                   str(self.password) + "@" +
23                                   str(self.server) + ":" +
24                                   str(self.port) +
25                                   "?authMechanism=DEFAULT&authSource=" +
26                                   str(self.db), serverSelectionTimeoutMS=500)
27         try:
28             if mongoClient.admin.command('ismaster')['ismaster']:
29                 return True, mongoClient
30
31         except OperationFailure:
32             return False, "Database not found."
33
34         except ServerSelectionTimeoutError:
35             return False, "MongoDB Server is down."
```

Ilustración 30

Por último, el directorio de archivos y carpetas del proyecto quedaría tal y como se muestra en la *Ilustración 31*.

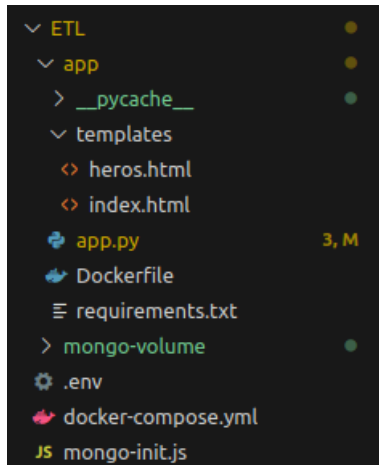


Ilustración 31

Dificultades encontradas

Durante la realización del proyecto se han encontrado múltiples dificultades, pero sin embargo, la que más tiempo consumió fue el no tener la versión correcta de Ubuntu y que esto resultaba altamente conflictivo con Docker a la hora de montar el servidor web Flask. Siempre, daba igual que repositorio se cogiera, resultaba con el error “http: Invalid Host Header”. Este error se solventó gracias a que se probó de instalar Docker desde otro ordenador y clonando los mismos repositorios con los que se estaba testeando y viendo que en el Mac sí que funcionaban. Así pues, esto llevó a la conclusión de que el problema era la máquina virtual, lo que llevó a más investigaciones hasta encontrar que se trataba de la versión de Ubuntu. Gracias al comando “sudo snap refresh docker --channel=latest/edge” se actualice, y se pudo empezar el proyecto.

Por otro lado, otras dificultades fueron las conexiones con la base de datos y el entendimiento general de la estructura que a medida que se iba avanzando en el proyecto se iban adquiriendo estos conocimientos.

Por último, el depurar ha sido un reto todo y que gracias a poderse conectar a los logs del servidor de Flask se podían ver cuáles eran los errores y donde se fallaba.

Añadir, que todo y que ChatGPT ha sido de gran ayuda para poder tener rápido acceso a por ejemplo las plantillas de código html y css, también se han usado los foros ya que muchas veces se tiene que ir más allá en la búsqueda.

Bibliografía

ChatGPT. (s. f.). <https://chat.openai.com/>

Docker-compose ERROR [internal] booting buildkit, http: invalid host header. (s. f.). Stack Overflow. <https://stackoverflow.com/questions/77225539/docker-compose-error-internal-booting-buildkit-http-invalid-host-header>

Flores, J. M. & F. (s. f.). *Python, Flask y MongoDB con Docker compose*. Python, Flask y MongoDB con Docker Compose. https://www.crashell.com/estudio/python_flask_y_mongodb_con_docker_compose

How to use Render_Template in Flask. (2023, 9 marzo). https://pytutorial.com/how-to-use-render_template-in-flask/?expand_article=1

Jarroba, R. [. (2017, 21 octubre). *Python MongoDB Driver, con ejemplos*. Jarroba. <https://jarroba.com/python-mongodb-driver-pymongo-con-ejemplos/>

Marvel Developer Portal. (s. f.). Marvel.com. <https://developer.marvel.com/>