Report (Microservices)

19.12.2022

Juan Domínguez
SPS
Chetumal, MX

Overview

SPS Microservices report solution using *Python*, a *NoSQL* database, and *containers*.

This is a second version of a solution presented a year ago: V1. The repository with the code for this new version can be found at V2.

Specifications

A. Programing language: *Python*
B. Database: *MongoDB*
C. Cloud: *fly.io*

Milestones

I. Python

We are using Python3, so we instantiated a local virtual environment where we would be able to install dependencies and keep our development clean. These steps are documented in the repo's *README.md* file.

II. API

For the main API, we choose to use the "Flaskr" example from *Flask* documentation as reference. It consists of a simple "posts" API, with *CRUD* capabilities which also provides simple authentication endpoints. However, we made some modifications in order to make it *RESTful* and fun.

Most of this process is documented in the repository history of commits, but we basically followed the next steps.

1. Setup the project directory in which our app would live.

2. Use the Flask app factory approach, using *create_app* as our entrypoint. Here the main *app* with its configurations and *blueprints* would be registered.

3. Then, we set up the database. As we choose *pymongo* as dependency, we can use it to get the connection object to manage collections and documents. At first we used a cloud stored instance (*MongoDB Atlas)* for local development.

4. Once we had access to the db, we followed with the authentication service to register and login users. JWT tokens have been added to this project.

5. With users being able to register, we continued with the blogs API, where *CRUD* operations were developed.

6. Throughout the previous steps, we added a custom error handler to standardize service responses.

III. Hypermedia

As we developed this service to return *JSON* responses, we decided to add a fun endpoint as an *Index* for the service, where you get a greeting from a random pokemon sprite rendered into the webpage to which you can select to be redirected to the main repository. This page uses an external API (*pokeapi.co*)

Welcome to Flaskr API

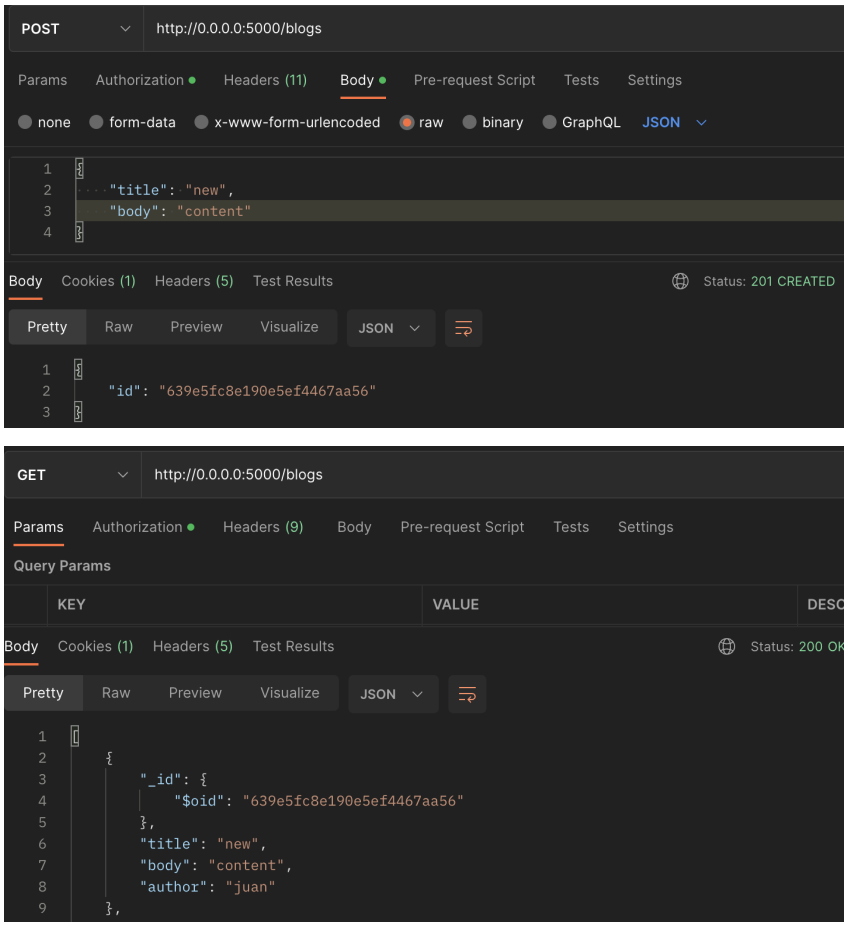
IV. Local

Once the basic project is completed, it can be deployed locally. These steps are documented in the README.md file.

```
(venv) →  flsk git:(master) ✗ flask --app flaskr --debug run
 * Serving Flask app 'flaskr'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a pro
duction WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 138-235-088
```

V. Postman

Using *postman* as a *REST* client, we test the registered endpoints.



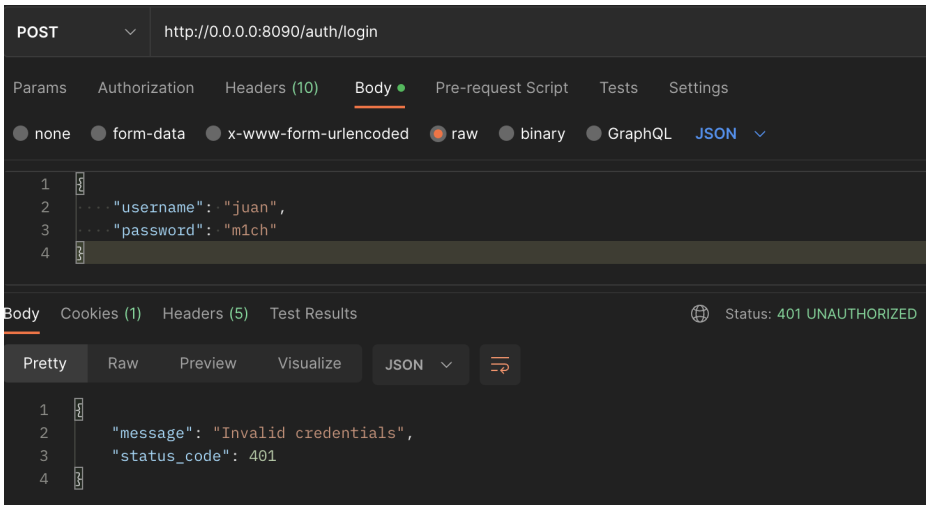
VI. Dockerize

For this stage, we needed to add a *Dockerfile*, where the default exposed port was modified. And after successfully building our image, we added a *docker-compose* file to use *MongDB* as a container too.

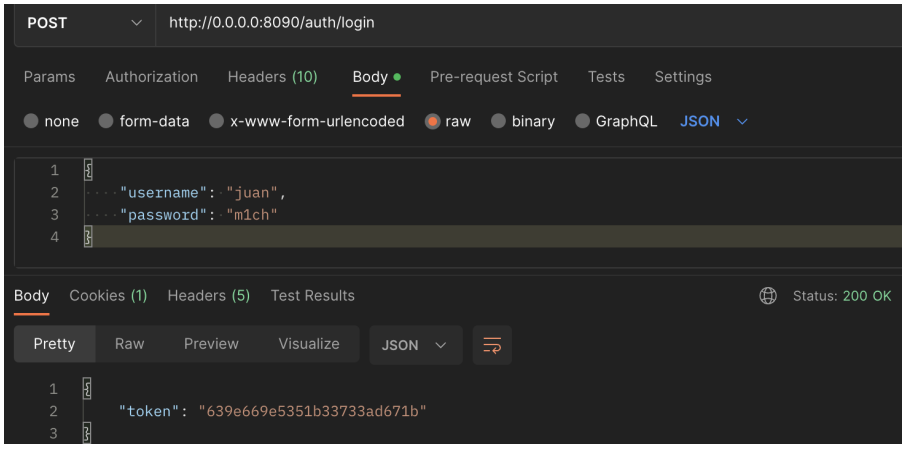
```
flsk-db-1       | {"t":{"$date":"2022-12-18T00:36:46.436+00:00"},"s":"I",  "c":"NETWORK",
 "id":23015,   "ctx":"listener","msg":"Listening on","attr":{"address":"0.0.0.0"}}
flsk-db-1       | {"t":{"$date":"2022-12-18T00:36:46.436+00:00"},"s":"I",  "c":"NETWORK",
 "id":23016,   "ctx":"listener","msg":"Waiting for connections","attr":{"port":27017,"ssl"
:"off"}}
flsk-backend-1  | 172.25.0.1 - - [18/Dec/2022 00:36:50] "GET / HTTP/1.1" 200 -
flsk-backend-1  | 172.25.0.1 - - [18/Dec/2022 00:36:50] "GET /static/style.css HTTP/1.1" 2
00 -
flsk-backend-1  | 172.25.0.1 - - [18/Dec/2022 00:36:50] "GET /static/27.png HTTP/1.1" 200
```

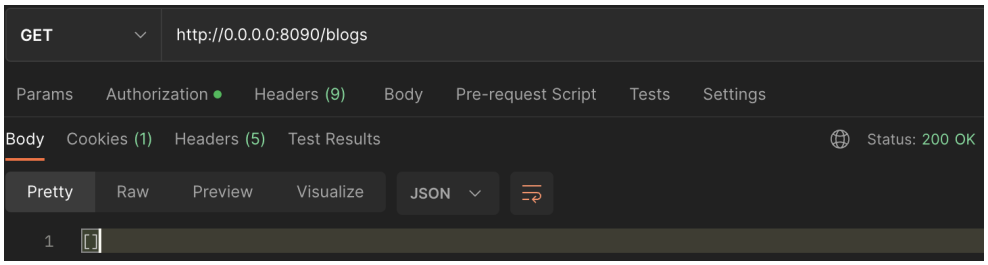
VII. Container

Using *postman* again, we repeated our tests but this time with the service running from the docker container orchestrated by the compose file. The next screenshot shows the new exposed port as we fail to login because the user doesn't exist in the contained database.



So, once registered, it got a different token response, and no posts were created yet either.





VIII. Minikube

For this task, we needed to install *minikube* locally in order to have a local cluster for testing. Then, a *deployment.yaml* file based on the dockerized image was written and added to the repository.

IX. Kubernetes

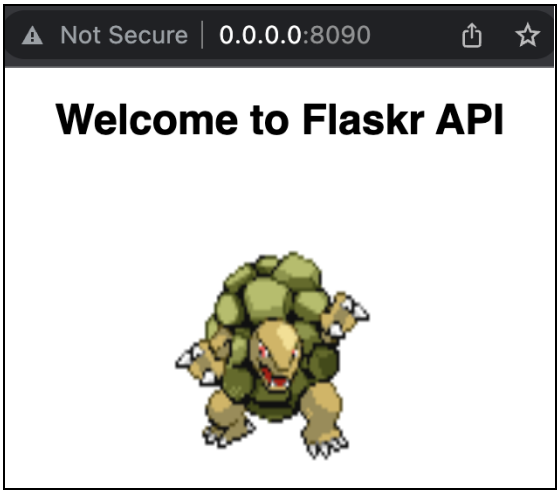
Now, we can deploy and use our services. But first we need to load the image into the *minikube* virtual machine. Then, using *kubectl* we apply our deployment manifest.

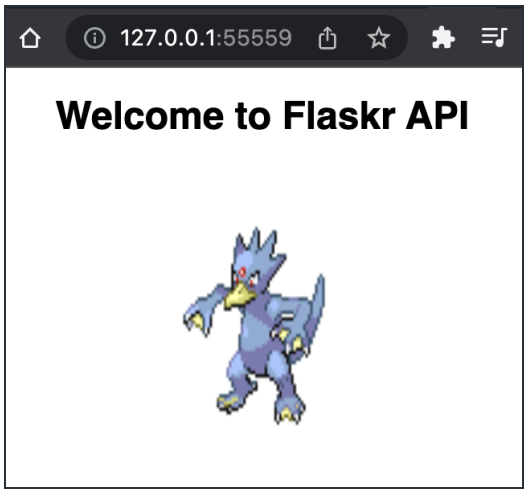
For this test we deployed three replicas.

```
(venv) ➜  flsk git:(master) ✗ kubectl get all
NAME                            READY   STATUS    RESTARTS   AGE
pod/flaskr-574d997d7c-l5s8s   1/1     Running   0          7s
pod/flaskr-574d997d7c-qrw8w   1/1     Running   0          7s
pod/flaskr-574d997d7c-xvxzv   1/1     Running   0          7s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
service/flaskr       ClusterIP   10.96.55.8   <none>        8090/TCP   7s
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP    2d4h

NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/flaskr   3/3     3            3           7s

NAME                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/flaskr-574d997d7c   3         3         3       7s
```

We can use port-forwarding to reach the service through a specific port, as mentioned in the *README.md* instructions. But we can also use *minikube* to expose the service.

```
(venv) ➜  flsk git:(master) ✗ minikube service flaskr
|-----------|---------|-------------|---------------|
| NAMESPACE |  NAME   | TARGET PORT |      URL      |
|-----------|---------|-------------|---------------|
| default   | flaskr  |             | No node port  |
|-----------|---------|-------------|---------------|
😿  service default/flaskr has no node port
🏃  Starting tunnel for service flaskr.
|-----------|---------|-------------|-------------------------|
| NAMESPACE |  NAME   | TARGET PORT |          URL            |
|-----------|---------|-------------|-------------------------|
| default   | flaskr  |             | http://127.0.0.1:55559  |
|-----------|---------|-------------|-------------------------|
```





X. Pods

This one was tricky, as we had to *"stop"* one pod and see what happens. This is not possible, but we can *scale* our deployment, as mentioned before, we had three replicas. So, we just scale it back to only two.

```
(venv) ➜  flsk git:(master) ✗ kubectl scale deployment flaskr --replicas=2
deployment.apps/flaskr scaled
(venv) ➜  flsk git:(master) ✗ kubectl get all
NAME                          READY   STATUS        RESTARTS   AGE
pod/flaskr-6b67bc6b96-7hzz6   1/1     Running       0          28m
pod/flaskr-6b67bc6b96-96dcp   1/1     Running       0          28m
pod/flaskr-6b67bc6b96-cx9wj   1/1     Terminating   0          28m

NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/flaskr       ClusterIP   10.101.130.157   <none>        8090/TCP   28m
service/kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP    2d5h

NAME                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/flaskr   2/2     2            2           28m

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/flaskr-6b67bc6b96   2         2         2       28m
```

XI. SaaS

For this step, we choose *fly.io*, as we had already used it before in *V1*, it's quite straightforward. Download the CLI service *flyctl* and use it to login and launch.

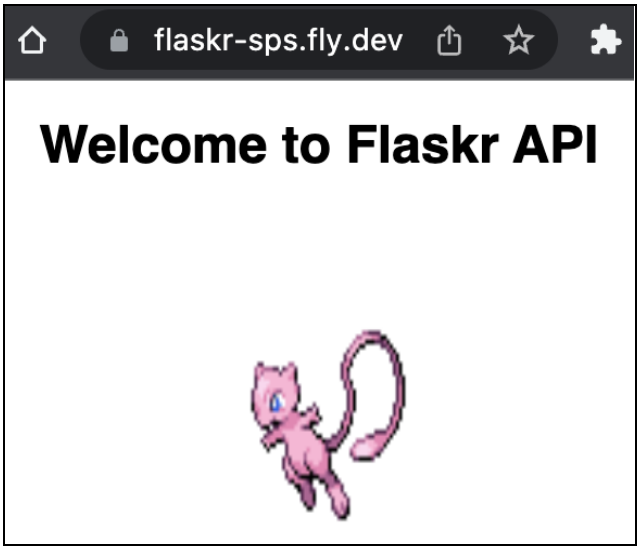
This service generates a *toml* file which then is used to deploy the app into the cloud. Although it uses the previously generated Dockerfile to do so, it is worth reviewing the file configurations and validating them.

Finally, for our database we needed to set up the stored Atlas DB URI as a secret variable via CLI.

```
--> Pushing image done
image: registry.fly.io/flaskr-sps:deployment-01GMM11KGV2TP844PVEJ7JT74K
image size: 155 MB
==> Creating release
--> release v3 created

--> You can detach the terminal anytime without stopping the deployment
==> Monitoring deployment
Logs: https://fly.io/apps/flaskr-sps/monitoring

 1 desired, 1 placed, 1 healthy, 0 unhealthy
--> v3 deployed successfully
```

XII. Cloud

Now, let's try it on the cloud.



XIII. Repo

The whole code is already on *github* and for each previous stage we can find a couple commits there.

XIV. Documentation

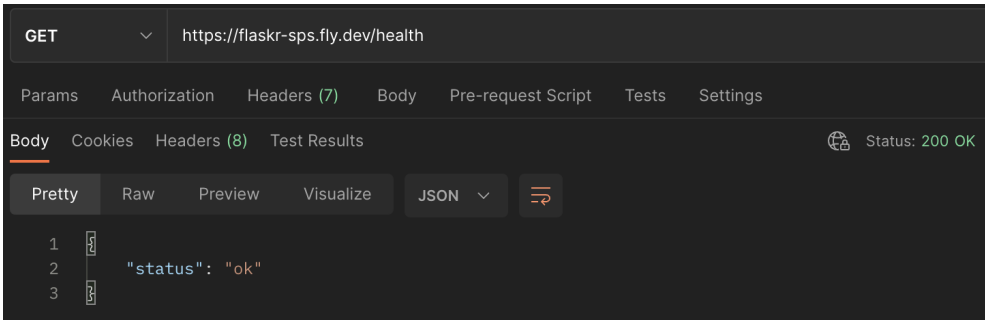
This report is also in the repo, inside the */docs* directory.

XV. Extras

As the project was growing, a couple points from this stage were taken into account.

Healthcheck

This endpoint was developed as simply as possible, it checks the db connection and sends a simple success response.



Docker Image

For building the image, we choose *python:3-slim* to make it lighter. We also used a *dockerignore* file and a “no cache” policy.

Documentation

For this task, we used *swagger*, to document our RESTful endpoints automatically form resource docstrings. This endpoint can be reached through the */spec* route which returns a JSON spec file as no Swagger UI was implemented yet, as it was too much of a hassle to implement for the time constraints we had, so it would be completed later.