

TD 7

Introduction

Nous avons à présent une architecture distribuée pour servir notre application. Nous allons maintenant nous attarder sur sa supervision. On cherche en effet à comprendre le fonctionnement de notre système à l'instant T et à émettre des alertes en fonction des événements.

Partie I – Emulation de clients

Nous allons émuler des clients se connectant à nos services. Ils réaliseront des opérations de lecture, ajout et suppression d'images.

1) On va utiliser « Puppeteer », un navigateur sans tête.

Avec la documentation de Puppeteer on va créer un service capturant notre galerie et la sauvegardant.

On crée un nouveau répertoire puppeteer. On installe puppeteer avec « npm i puppeteer ». On fait un « npm init » puis on remplit un fichier Dockerfile.

```
FROM alekzonder/puppeteer:latest
COPY . /app
WORKDIR app

CMD [ "npm", "start" ]
```

On ajoute le code ci-dessous en suivant la doc de puppeteer :

```
docker-compose.yml | puppeteer.js
const puppeteer = require('puppeteer');

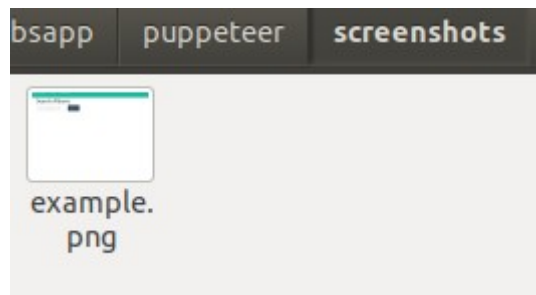
(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto('http://12.0.0.2:3000/');
  await page.screenshot({path: './screenshots/example.png'});

  await browser.close();
})();
```

On se rend sur le répertoire contenant le fichier puppeteer.js puis on l'exécute :

```
orain@orain-TM1701:~/Documents/td7-antho35/td7/3VM/myhbsapp/puppeteer$ node puppeteer.js
```

Une capture de notre application (avec l'adresse IP de notre manager) a bien été créée dans mon dossier screenshots :



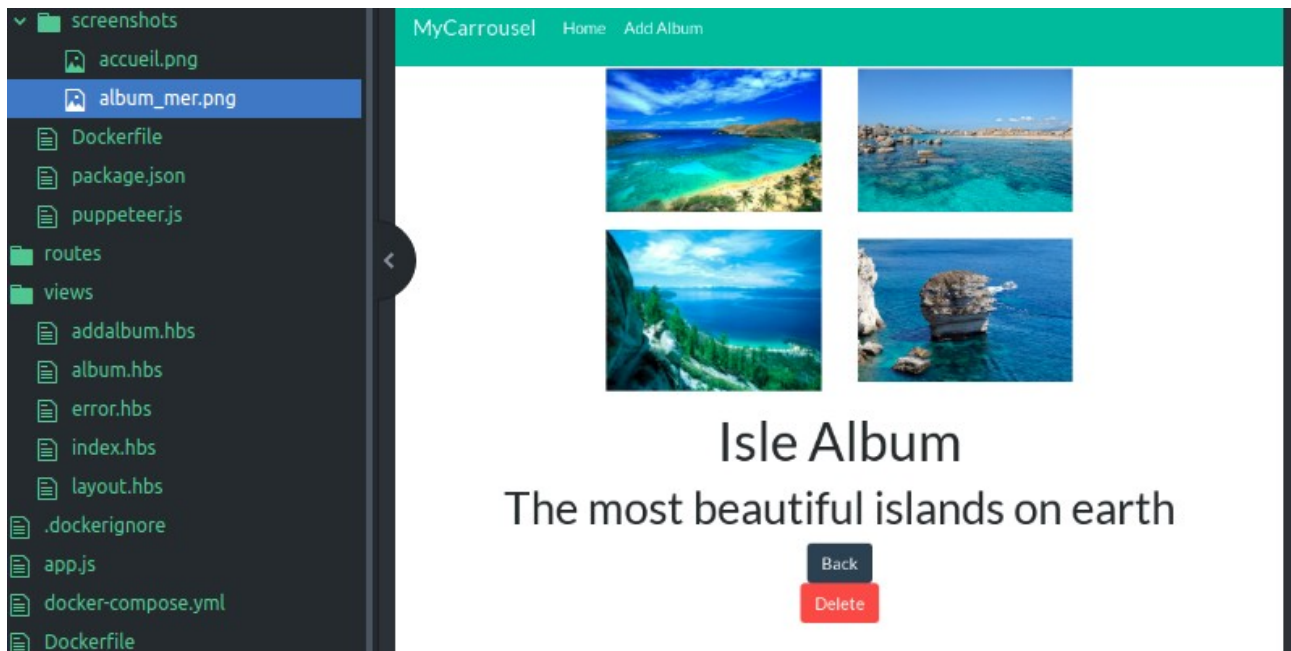
2)

Nous allons simuler un utilisateur qui interagit avec notre application.

Lorsque l'on arrive sur mon application. Nous avons un formulaire qui permet de rechercher un album. Dans le code ci-dessous, après avoir demandé de prendre une capture de la page d'accueil, je simule qu'on entre la chaîne « album_mer » dans mon input, puis je clique sur le bouton de recherche. Je prends une capture à la suite de cela.

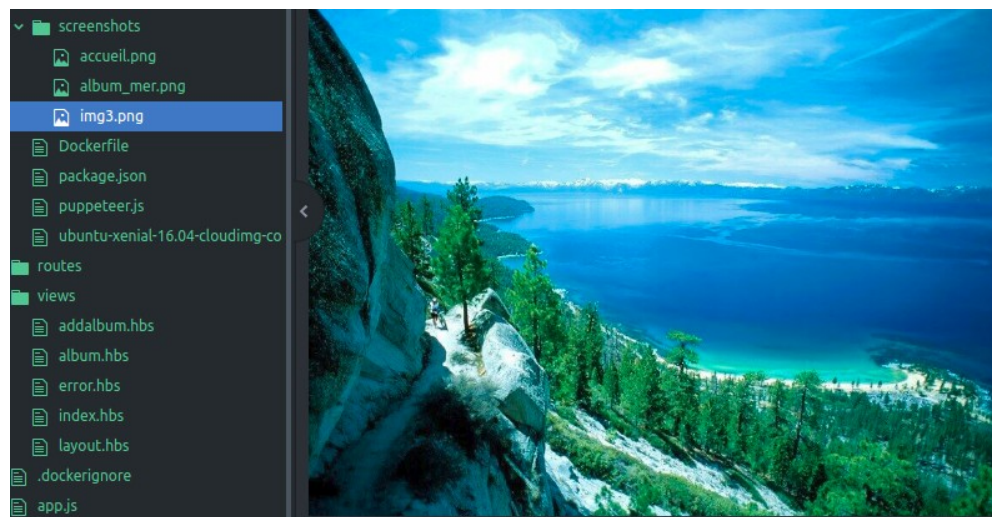
```
docker-compos... | puppeteer.js | accueil.png | index.hbs | addalbum.hbs
1  const puppeteer = require('puppeteer');
2
3  (async () => {
4    const browser = await puppeteer.launch();
5    const page = await browser.newPage();
6    await page.goto('http://12.0.0.2:3000/');
7    await page.screenshot({path: './screenshots/accueil.png'});
8
9    await page.type('input.form-control', 'album_mer');
10
11    await Promise.all([
12      page.waitForNavigation(),
13      page.click('input.btn.btn-primary')
14    ]);
15    await page.screenshot({path: './screenshots/album_mer.png'});
16    await browser.close();
17  })();
18
```

J'obtiens bien 2 captures d'écrans dans mon dossier screenshots.



Pour la suite, on souhaite que l'utilisateur clique sur l'image 3 :

```
await page.click('#img3');  
await page.screenshot({path: './screenshots/img3.png'});
```



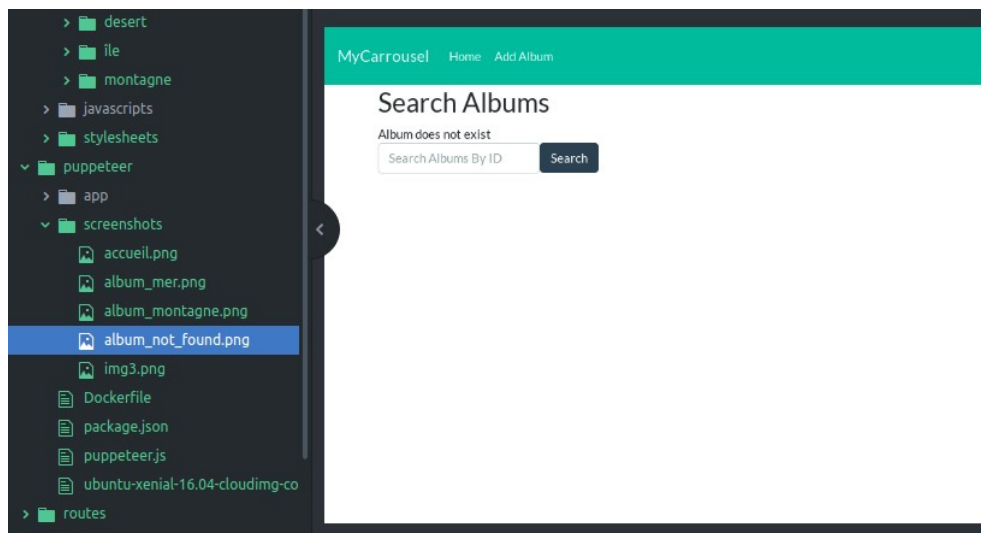
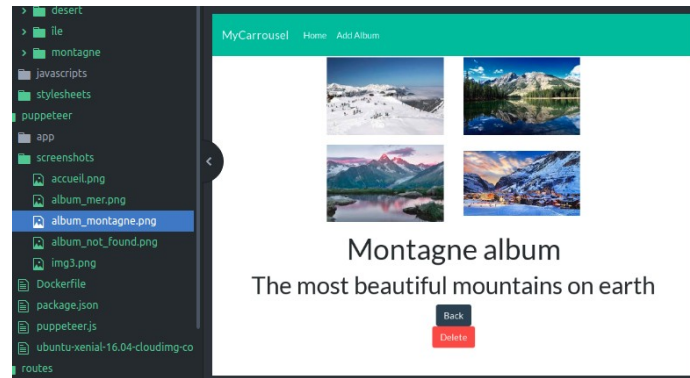
img3.png est ajoutée au dossier « screenshots ».

3)

Nous allons à présent émuler un utilisateur qui va supprimer un album.

Dans le code ci-dessous, l'utilisateur se rend à l'accueil, il cherche l'album « album_montagne ». Il clique sur « delete » pour supprimer l'album. Il est alors redirigé vers l'accueil. Il cherche à nouveau le même album mais il a le message suivant : « Album does not exist ». L'album a bien été supprimé.

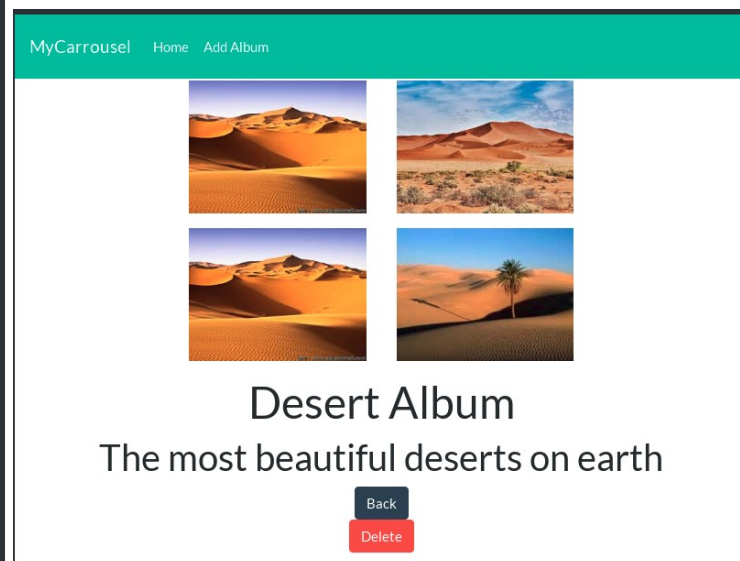
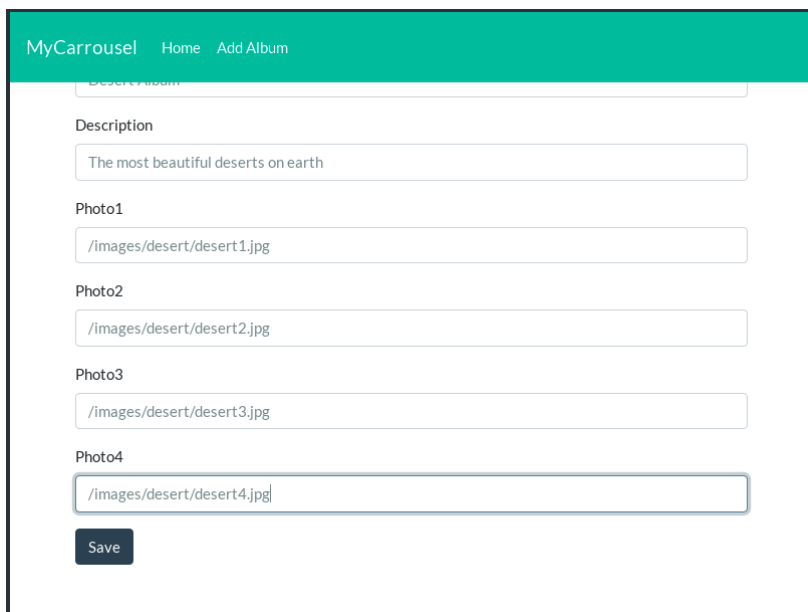
```
//l'utilisateur retourne à la page d'accueil
await page.goto('http://12.0.0.2:3000/');
//il choisit un autre album
await page.type('#searchalbum', 'album_montagne');
await Promise.all([
  page.waitForNavigation(),
  page.click('#btnsearch')
]);
await page.screenshot({path: './screenshots/album_montagne.png'});
await page.click('#delete');
await page.waitForNavigation();
await page.type('#searchalbum', 'album_montagne');
await Promise.all([
  page.waitForNavigation(),
  page.click('#btnsearch')
]);
await page.screenshot({path: './screenshots/album_not_found.png'});
```



4) Nous allons créer un utilisateur qui va ajouter un album.

Dans le code ci-dessous, on clique sur « Add Album » dans la barre du menu. On remplit les champs puis on clique sur le bouton pour envoyer le formulaire. On a alors deux captures : une en ayant rempli le formulaire et une autre où on a le résultat de la recherche « album_desert ».

```
await page.click('#addalbum');
await page.type('#id', 'album_desert');
await page.type('#titre', 'Desert Album');
await page.type('#description', 'The most beautiful deserts on earth');
await page.type('#photo1', '/images/desert/desert1.jpg');
await page.type('#photo2', '/images/desert/desert2.jpg');
await page.type('#photo3', '/images/desert/desert3.jpg');
await page.type('#photo4', '/images/desert/desert4.jpg');
await page.screenshot({path: './screenshots/addalbum_desert.png'});
await Promise.all([
  page.waitForNavigation(),
  page.click('#submit')
]);
await page.type('#searchalbum', 'album_desert');
await Promise.all([
  page.waitForNavigation(),
  page.click('#btnsearch')
]);
await page.screenshot({path: './screenshots/album_desert.png'});
```



5)

Pour les questions de probabilité il suffit de créer une fonction donnant un nombre entier compris entre 1 et 10 (soit entre 1 et 11 exclu). Ceci grâce à une fonction `math.random()` :

```
//proba
function getRandomArbitrary(min, max) { //valeur de max est exclue
  return Math.random() * (max - min) + min;
}
```

Pour l'ajout et la suppression d'un album avec une probabilité de 1/10, il faut créer la condition suivante :

```
if(getRandomArbitrary(1, 11)==1){
```

Dans cette condition, on ajoute le code correspondant à l'ajout ou à la suppression d'un album.

Pour l'utilisateur qui se promène avec une probabilité de 8/10, on ajoute la condition suivante :

```
if(getRandomArbitrary(1, 11)!=1 || getRandomArbitrary(1, 11)!=2){
```

Puis on met le code correspondant.

6)

Avant d'ajouter mon service au swarm, j'ai modifié mon service puppeteer dans le docker-compose.yml pour qu'il soit lié à mon application et en créant un volume du dossier correspondant :

```
puppeteer:
  image: 127.0.0.1:5000/puppeteer
  build: ./puppeteer
  ports:
    - "4000:4000"
  volumes:
    - "./puppeteer/screenshots:/screenshots"
```

On peut ensuite entrer la commande « sudo docker-compose up -d » puis un « sudo docker-compose push ».

On peut alors deploy le stack. Le service puppeteer a bien été ajouté au swarm.

```
vagrant@manager:/vagrant/myhsapp$ sudo docker stack services myhsapp
ID                NAME                MODE                REPLICAS            IMAGE                PORTS
l45pwmq3w392     myhsapp_web         replicated          1/1                 127.0.0.1:5000/myhsapp:latest    *:3000->3000/tcp
pm89jepofc7j     myhsapp_minio       replicated          1/1                 minio/minio:latest             *:9000->9000/tcp
rvbccrh6tg18     myhsapp_puppeteer   replicated          1/1                 127.0.0.1:5000/puppeteer:latest  *:4000->4000/tcp
ynx7nx4ldjlu     myhsapp_redis       replicated          1/1                 redis:alpine
```

Partie II – Monitoring avec Prometheus

Prometheus propose une solution de monitoring s'interfaçant avec Docker. L'objectif va être de l'installer sur notre swarm.

1) Nous devons créer un fichier daemon.json (vu qu'il n'existe pas) dans le répertoire /etc/docker/ à l'aide de Ansible. Dans notre manager_playbook.yml, on ajoute les lignes suivantes :

```
- name: create file daemon.json
  copy:
    dest: "/etc/docker/daemon.json"
    content: |
      {
        "metrics-addr" : "0.0.0.0:9323",
        "experimental" : true
      }
```

On peut appliquer les modifications de mon manager_playbook.yml en lançant : « vagrant up --provision ».

On démarre donc notre VM manager puis on regarde si le fichier daemon.json a bien été créé :

```
orain@orain-TM1701:~/Documents/td7-antho35/td7/3VM/myhbsapp$ vagrant ssh manager
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-141-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.

New release '18.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Wed Jan 30 15:41:16 2019 from 10.0.2.2
vagrant@manager:~$ cd /etc/docker
vagrant@manager:/etc/docker$ ls
daemon.json  key.json
```

2) On peut tester le bon fonctionnement des métriques :

```
vagrant@manager:/etc/docker$ curl http://localhost:9323/metrics
# HELP builder_builds_failed_total Number of failed image builds
# TYPE builder_builds_failed_total counter
builder_builds_failed_total{reason="build_canceled"} 0
builder_builds_failed_total{reason="build_target_not_reachable_error"} 0
builder_builds_failed_total{reason="command_not_supported_error"} 0
builder_builds_failed_total{reason="dockerfile_empty_error"} 0
builder_builds_failed_total{reason="dockerfile_syntax_error"} 0
builder_builds_failed_total{reason="error_processing_commands_error"} 0
builder_builds_failed_total{reason="missing_onbuild_arguments_error"} 0
builder_builds_failed_total{reason="unknown_instruction_error"} 0
# HELP builder_builds_triggered_total Number of triggered image builds
```

Il est important de surveiller les métriques d'une VM afin de veiller à sa performance. En particulier au niveau du CPU, le CPU Ready ne doit pas dépasser 5 %.

En effet, les goulots d'étranglement d'un VM conduisent à des ralentissements importants. On pourra configurer par la suite des alertes si on souhaite être prévenu pour un évènement donné.

3)

On ajoute un fichier de configuration dans le répertoire : /tmp

```
vagrant@manager:/tmp$ touch prometheus.yml
vagrant@manager:/tmp$ ls
prometheus.yml
vagrant@manager:/tmp$ nano prometheus.yml
```

```
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ['12.0.0.2:9090']

  - job_name: 'docker'
    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ['12.0.0.2:9323']
```

Dans la figure ci-dessus, on veille à bien indiquer l'adresse Ip de notre manager.

Avant de créer notre service, on redémarre notre serveur docker afin que les modifications faites dans daemon.json soient bien prises en compte.

Puis on crée le service :

```
vagrant@manager:/$ sudo service docker restart
vagrant@manager:/$ sudo docker service create --replicas 1 --name my-prometheus --mount type=bind,source=/tmp/prometheus.yml,destination=/etc/prometheus/prometheus.yml --publish published=9090,target=9090,protocol=tcp prom/prometheus
i20xv03nhniwh8y7hkxdin7g5
overall progress: 1 out of 1 tasks
1/1: running [=====]
verify: Service converged
vagrant@manager:/$
```

On peut à présent entrer dans notre navigateur l'adresse : <http://12.0.0.2:9090/targets>.

Prometheus Alerts Graph Status ▾ Help					
Targets					
All		Unhealthy			
docker (1/1 up)		show less			
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://12.0.0.2:9323/metrics	UP	instance="12.0.0.2:9323" job="docker"	12.64s ago	34.37ms	
prometheus (1/1 up)		show less			
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://12.0.0.2:9090/metrics	UP	instance="12.0.0.2:9090" job="prometheus"	2.955s ago	2.946ms	

On obtient bien le résultat souhaité.

Conclusion

Pour résumer, dans ce TP, j'ai pu apprendre à émuler un client afin de simuler une interaction avec mon application web. J'ai ainsi créé un client qui recherche des albums, clique sur des images, ajoutet et supprime des albums. Dans un second temps, j'ai mis en place une solution de monitoring s'interfaçant avec docker : Prometheus. Elle permet de comprendre comment fonctionne le système à un instant T. Nous allons nous intéresser par la suite aux alertes et visualisation de ces évènements.