# Labo HTTP Infra

## Directives summary

### Objectives

- Learn (web infra, apache2 and express.js)
- Implement (dynamic web app HTML, CSS, JS + Ajax Requests)
- Practice (docker)

### General instructions

- Instructions are given through videos at each step (if correctly done, ensure a grade of 4.5)
- The rest of the points come from your own research and creativity.
- We can use other technologies if we want (apache ⇔ nginx, express.js ⇔ django, ...)

  > Go ahead, we **LOVE** that

Nb: Provide a way to see the result of each step individually (?)

## Required Steps (max grade: 4.5)

## Step 1: Static HTTP server with apache httpd

### Goals

- Create a github repository
- Create a apache2 docker image with custom content

startbootstrap.com: some bootstrap templates.

The template we used: Freelancer (download)

```
# Build
## Using Docker
docker build -f apache2.Dockerfile -t res-http-apache2-static .
## Using Docker compose
docker-compose build

# Run
## Using Docker
docker run -p "8080:80" res-http-apache2-static
## Using Docker compose (Build + Run)
docker-compose up # add -d option to run as a daemon (i.e in the background)
```

## Step 2: Dynamic HTTP server with express.js

### Goals

- Write a dynamic HTTP app (express.js)
- Query the server (postman)

We made 3 versions:

- ExpressJs
- Flask
- CrowCpp: The build has been leveraged using 2 methods:
  - Using a docker container as a build environement:

    ```
    # Build the build environment image
    docker build -f build.Dockerfile -t res-crow-build .
    # Mount the sources and build. The binary will then be available in the
    sources' folder
    docker run --rm -v "$PWD:/build" res-crow-build g++ server.cpp -o server
    -lpthread
    # Create the final image by copying the binary inside of it
    docker build --no-cache -f crow.Dockerfile -t res-crow .
    ```

    (see `step2/cpp/build.sh` script)

    This method is better when building with local cache (e.g. node) since we won't have to pull then everytime.

  - Using Docker multi-step build:

    1. One image is created with the required package to build
    2. A second image is created from the previous one and the sources. The binary is built inside of this image
    3. This final image will simply copy from the second one the compiled binary.

    This method is standalone and perfectly reproducible, but will take longueur since it won't be able to remember cache information between the builds

## Step 3: Reverse proxy with apache (static configuration)

### Goals

see Reverse Proxy Guide

### Remarks

- This step re-use the images generated on step 1 and 2. They must have been built beforehand.
- We used `*.localhost` domains to avoid dealing with DNS and updating configuration files.
- This is NOT possible to prevent access to containers from host by just using Docker. The containers are using interfaces on the host machine. **BUT** the browsers have a same-origin-policy which prevent cross-origin-resource-sharing, i.e. fetch data from another source than the current page's one.
  But, on Windows and Mac using Docker-Desktop, docker is run in a virtual machine. In this case, the services are not available from the host directly without using a port forward.
- Docker networks have their own dns resolution, we do not need to use static ip adresses and can use hostnames instead. This allow us to have 2 or more proxypass for the same host using `aliases` to have multiple domains for each host.
- The apache static configuration will have to be updated manually each time a network change is made (change of ip/hostname, adding/removing service/replicas, ...)

## Step 4: AJAX requests with JQuery

### Goals

- Use Jquery to make an AJAX request

- Jquery is not part of bootstrap anymore. We had to import it from a CDN.

  ```
  <script src="https://code.jquery.com/jquery-3.6.0.min.js"
  integrity="sha256-/xUj+3OJU5yExlq6GSYGSHk7tPXikynS7ogEvDej/m4="
  crossorigin="anonymous"></script>
  ```

### Remarks

- This step re-use the images generated on step 1 and 2. They must have been built beforehand.

## Step 5: Dynamic reverse proxy configuration

### Goals

- Use traefik for dynamic reverse proxy.

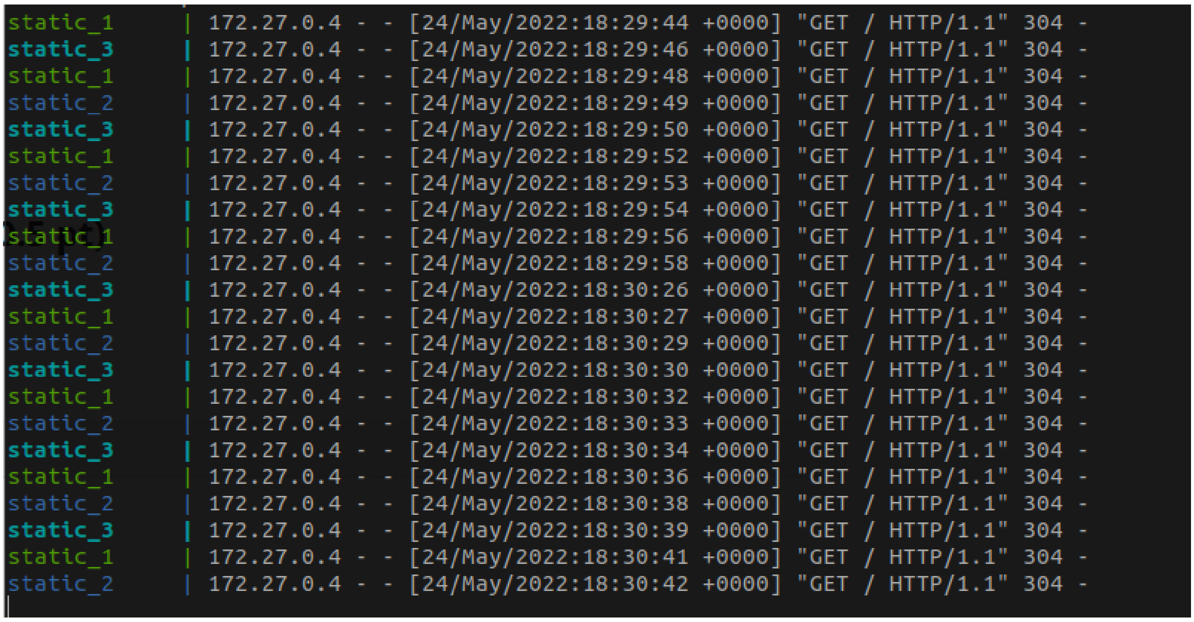
# Additional steps to get extra points on top of the "base" grade

## Load balancing: multiple server nodes (0.5 pt)

Using Traefik, we juste need to have many instance of the same service routed by traefik (i.e. the correct labels must be defined)

```
- "traefik.enable=true"
- "traefik.http.routers.static.rule=Host(`res-http.localhost`)"
- "traefik.http.routers.static.entrypoints=web"
```

We can see that the default behavior is using a round-robin load balancing.



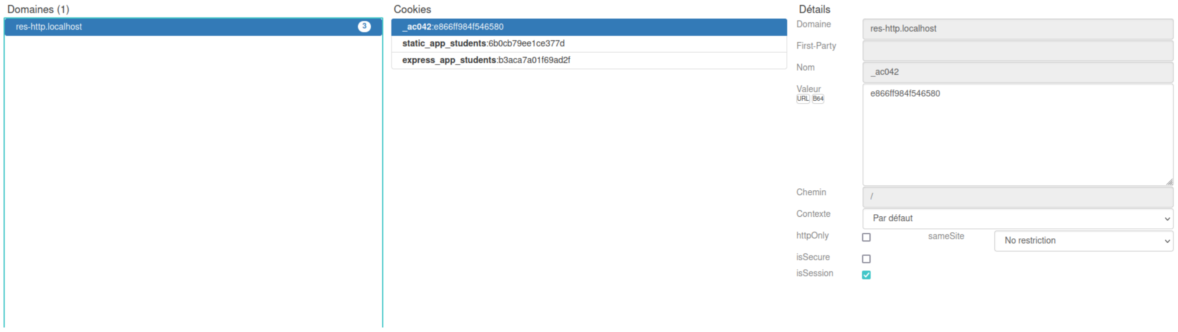
```
static_1      | 172.27.0.4 - - [24/May/2022:18:29:44 +0000] "GET / HTTP/1.1" 304 -
static_3      | 172.27.0.4 - - [24/May/2022:18:29:46 +0000] "GET / HTTP/1.1" 304 -
static_1      | 172.27.0.4 - - [24/May/2022:18:29:48 +0000] "GET / HTTP/1.1" 304 -
static_2      | 172.27.0.4 - - [24/May/2022:18:29:49 +0000] "GET / HTTP/1.1" 304 -
static_3      | 172.27.0.4 - - [24/May/2022:18:29:50 +0000] "GET / HTTP/1.1" 304 -
static_1      | 172.27.0.4 - - [24/May/2022:18:29:52 +0000] "GET / HTTP/1.1" 304 -
static_2      | 172.27.0.4 - - [24/May/2022:18:29:53 +0000] "GET / HTTP/1.1" 304 -
static_3      | 172.27.0.4 - - [24/May/2022:18:29:54 +0000] "GET / HTTP/1.1" 304 -
static_1      | 172.27.0.4 - - [24/May/2022:18:29:56 +0000] "GET / HTTP/1.1" 304 -
static_2      | 172.27.0.4 - - [24/May/2022:18:29:58 +0000] "GET / HTTP/1.1" 304 -
static_3      | 172.27.0.4 - - [24/May/2022:18:30:26 +0000] "GET / HTTP/1.1" 304 -
static_1      | 172.27.0.4 - - [24/May/2022:18:30:27 +0000] "GET / HTTP/1.1" 304 -
static_2      | 172.27.0.4 - - [24/May/2022:18:30:29 +0000] "GET / HTTP/1.1" 304 -
static_3      | 172.27.0.4 - - [24/May/2022:18:30:30 +0000] "GET / HTTP/1.1" 304 -
static_1      | 172.27.0.4 - - [24/May/2022:18:30:32 +0000] "GET / HTTP/1.1" 304 -
static_2      | 172.27.0.4 - - [24/May/2022:18:30:33 +0000] "GET / HTTP/1.1" 304 -
static_3      | 172.27.0.4 - - [24/May/2022:18:30:34 +0000] "GET / HTTP/1.1" 304 -
static_1      | 172.27.0.4 - - [24/May/2022:18:30:36 +0000] "GET / HTTP/1.1" 304 -
static_2      | 172.27.0.4 - - [24/May/2022:18:30:38 +0000] "GET / HTTP/1.1" 304 -
static_3      | 172.27.0.4 - - [24/May/2022:18:30:39 +0000] "GET / HTTP/1.1" 304 -
static_1      | 172.27.0.4 - - [24/May/2022:18:30:41 +0000] "GET / HTTP/1.1" 304 -
static_2      | 172.27.0.4 - - [24/May/2022:18:30:42 +0000] "GET / HTTP/1.1" 304 -
```

## Load balancing: round-robin vs sticky sessions (0.5 pt)

To use sticky sessions, we need 2 labels:

```
- "traefik.http.services.myservername.loadbalancer.sticky.cookie=true"
- "traefik.http.services.myservername.loadBalancer.sticky.cookie.name=myservice_cookie_name"
```

- the `cookie=true` enable the sticky session cookie
- the `cookie.name=...` used to know the cookie used to remember the destination server to use.
  By default, a name is provided using a hash



Domaines (1)

res-http.localhost 3

Cookies

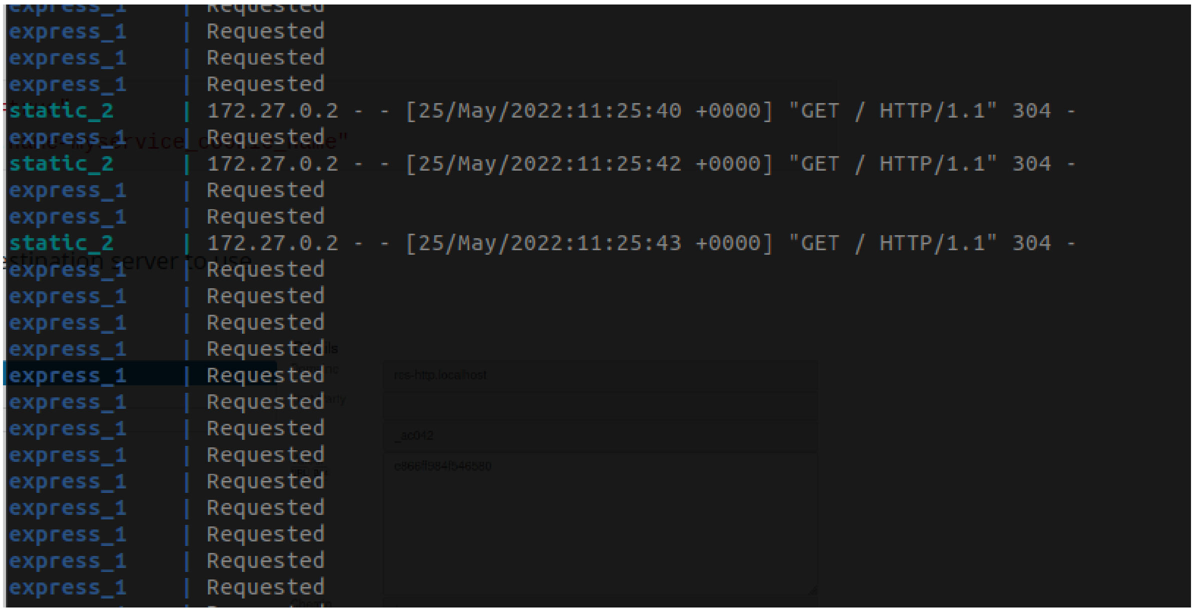
_ac042:e866ff984f546580
static_app_students:6b0cb79ee1ce377d
express_app_students:b3aca7a01f69ad2f

Détails

Domaine: res-http.localhost
First-Party:
Nom: _ac042
Valeur: e866ff984f546580
Chemin: /
Contexte: Par défaut
httpOnly ☐ sameSite: No restriction
isSecure ☐
isSession ☑

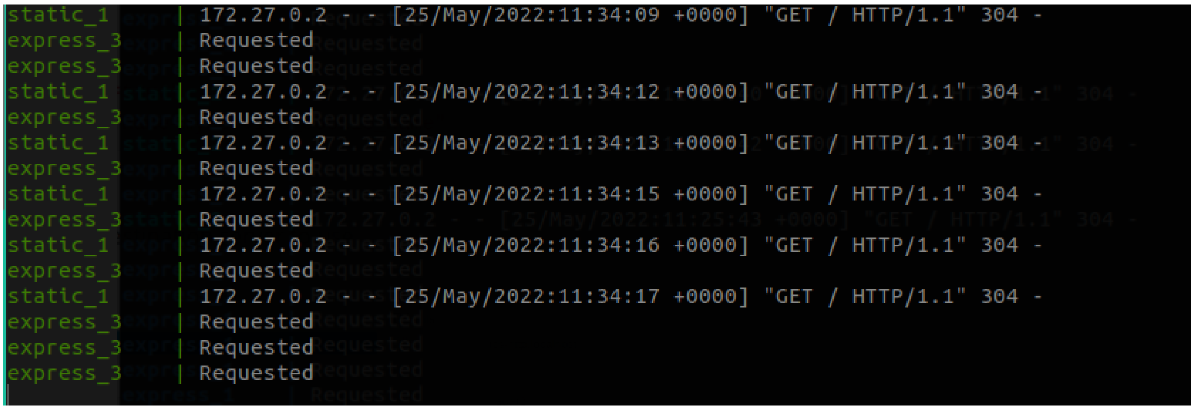
(This view of the cookie is provided by "Cookie Quick Manager" Firefox extension)

As we can see, each reload of the page will request the `static_2` server, eventhough there are 2 other server `statcic_1` and `static_3`. Also, this server will always query the same express server `express_1`



```
express_1    | Requested
express_1    | Requested
express_1    | Requested
express_1    | Requested
static_2     | 172.27.0.2 - - [25/May/2022:11:25:40 +0000] "GET / HTTP/1.1" 304 -
express_1    | Requested
static_2     | 172.27.0.2 - - [25/May/2022:11:25:42 +0000] "GET / HTTP/1.1" 304 -
express_1    | Requested
express_1    | Requested
static_2     | 172.27.0.2 - - [25/May/2022:11:25:43 +0000] "GET / HTTP/1.1" 304 -
express_1    | Requested
express_1    | Requested
express_1    | Requested
express_1    | Requested
express_1    | Requested
express_1    | Requested
express_1    | Requested
express_1    | Requested
express_1    | Requested
express_1    | Requested
express_1    | Requested
express_1    | Requested
express_1    | Requested
```

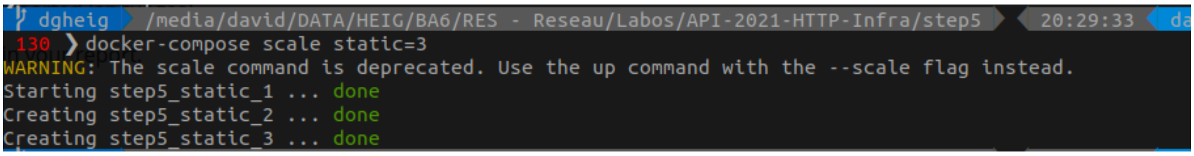
But when accessed from another context (using another browser or private navigation), we may use other servers



```
static_1    | 172.27.0.2 - - [25/May/2022:11:34:09 +0000] "GET / HTTP/1.1" 304 -
express_3   | Requested
express_3   | Requested
static_1    | 172.27.0.2 - - [25/May/2022:11:34:12 +0000] "GET / HTTP/1.1" 304 -
express_3   | Requested
static_1    | 172.27.0.2 - - [25/May/2022:11:34:13 +0000] "GET / HTTP/1.1" 304 -
express_3   | Requested
static_1    | 172.27.0.2 - - [25/May/2022:11:34:15 +0000] "GET / HTTP/1.1" 304 -
express_3   | Requested
static_1    | 172.27.0.2 - - [25/May/2022:11:34:16 +0000] "GET / HTTP/1.1" 304 -
express_3   | Requested
static_1    | 172.27.0.2 - - [25/May/2022:11:34:17 +0000] "GET / HTTP/1.1" 304 -
express_3   | Requested
express_3   | Requested
express_3   | Requested
```

## Dynamic cluster management (0.5 pt)

```
docker-compose scale static=3
```



```
 dgheig  /media/david/DATA/HEIG/BA6/RES - Reseau/Labos/API-2021-HTTP-Infra/step5   20:29:33  da
 130 ❯ docker-compose scale static=3
WARNING: The scale command is deprecated. Use the up command with the --scale flag instead.
Starting step5_static_1 ... done
Creating step5_static_2 ... done
Creating step5_static_3 ... done
```

This command will take the service `static` (according to the docker-compose file) and create/delete instances to match the requested number of instance (here: 3).

We can scale many services at once

```
docker-compose scale static=3 express=4
```

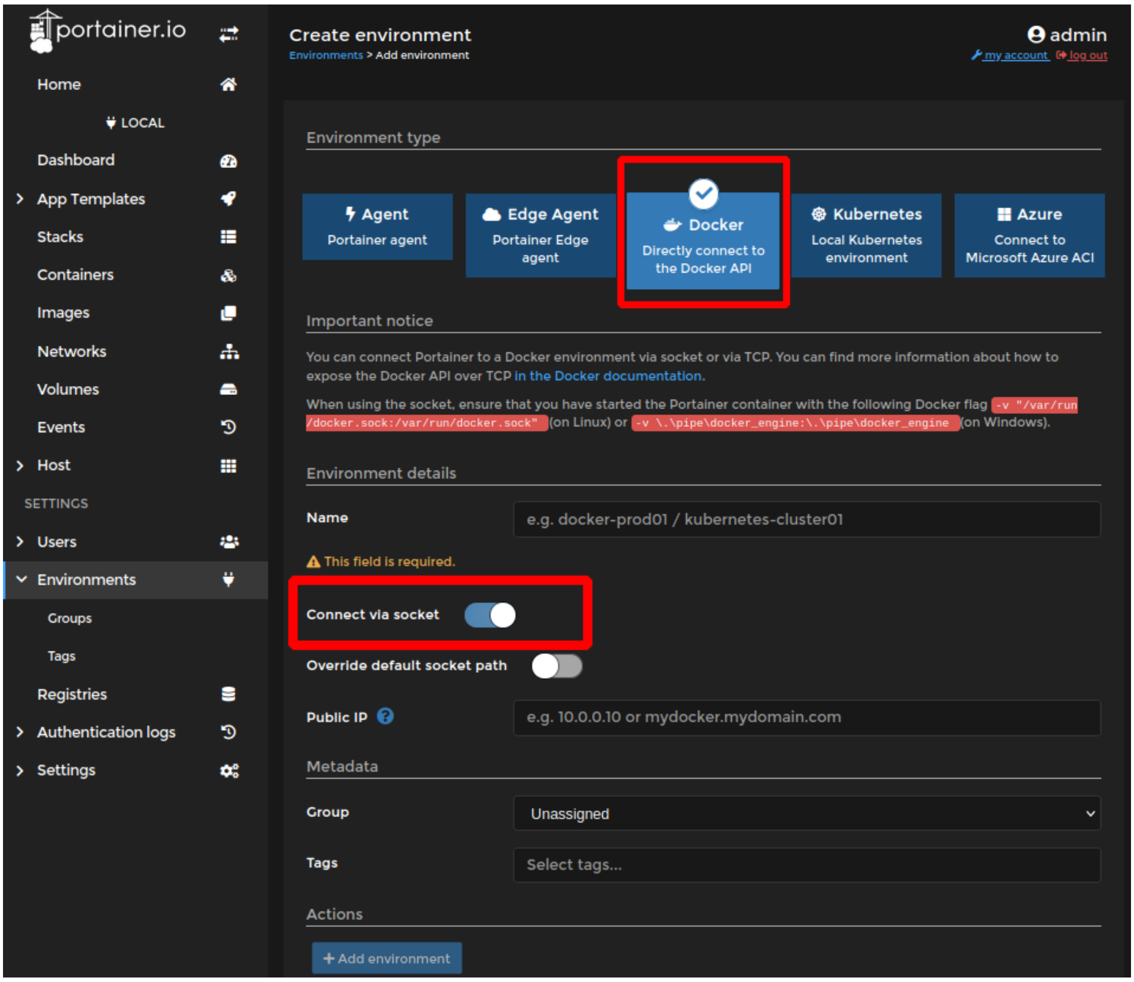
## Management UI (0.5 pt)

We will use Portainer-ce.

```
  portainer:
    image: portainer/portainer-ce:latest
    container_name: portainer
    restart: unless-stopped
    security_opt:
      - no-new-privileges:true
    volumes:
      - /etc/localtime:/etc/localtime:ro
      - /var/run/docker.sock:/var/run/docker.sock:ro
      - portainer-data:/data
    ports:
      - 9000:9000
```

Nb: We do not need special routing from traefik.

- The UI is available at `localhost:9000`.
- We need to configure a local docker by using the socket (available inside the container through a mount)



Environment type

Important notice

You can connect Portainer to a Docker environment via socket or via TCP. You can find more information about how to expose the Docker API over TCP in the Docker documentation.

When using the socket, ensure that you have started the Portainer container with the following Docker flag `-v "/var/run/docker.sock:/var/run/docker.sock"` (on Linux) or `-v \.\pipe\docker_engine:\.\pipe\docker_engine` (on Windows).

Environment details

Name: e.g. docker-prod01 / kubernetes-cluster01

⚠ This field is required.

Connect via socket ●

Override default socket path ○

Public IP: e.g. 10.0.0.10 or mydocker.mydomain.com

Metadata

Group: Unassigned

Tags: Select tags...

Actions