HTTPINFRA

Laboratoire RES

Rémi Jacquemard & Aurélie Lévy

Table des matières

Introduction

Durant 6 semaines nous avons dû faire un laboratoire assez long. Ce dernier concernait l'apprentissage de l'infrastructure http, avec utilisation d'un serveur Apache et de la technologie Docker. Le but principal était de créer une application web dynamique, en utilisant entre autres du HTML, du CSS, du Javascript et des requêtes AJAX.

Etape 1 : serveur http statique avec apache httpd

Branche GIT

Pour cette première étape, vous pouvez trouver le résultat final sur la branche **fb-apache-static**.

Structure finale

Httpd signifie Hypertext Transfer Protocol Daemon. Il s'agit d'un logiciel tournant en arrière-plan d'un serveur web et qui attend les requêtes du serveur.

La structure finale est constituée de deux containers, chacun à l'écoute sur le port standard http (80), possédant chacun un fichier de configuration et des fichiers de contenu. Nous pouvons, en utilisant le protocole http et le port mapping, échanger avec ces containers en passant par la VM.

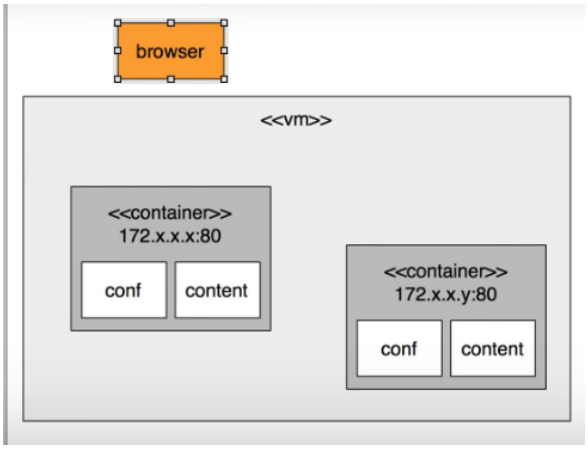

Figure 1 Structure générale étape 1

Pour commencer, une fois notre repo créé, il nous a fallu trouver une image Docker fournissant un serveur Apache. Pour cela, nous sommes allés sur le Docker Hub et avons téléchargé l'image officiel PHP (php official), laquelle fourni un serveur Apache avec gestion du PHP. En effet, cette image est déjà configurée pour pouvoir utiliser php avec le serveur.

Dockerfile

Afin de faire la configuration de l'image, nous avons modifié le fichier Dockerfile et avons mis deux lignes importantes :

```
FROM php:7.0-apache
COPY src/ /var/www/html/
```

La ligne commençant par FROM permet d'annoncer à l'image quelle version nous utilisons. Ici, il s'agit de la version 7.0 d'Apache.

En ce qui concerne la deuxième ligne (COPY), cela permet de demander la copie du dossier *src/* de notre file system local dans le dossier */var/www/html/* du file system de l'image docker. Cela permet

de pouvoir redémarrer un container et de pouvoir directement utiliser les fichiers/dossiers gardés en local qui, s'ils étaient uniquement dans le container, seraient effacés à l'extinction de ce dernier. Ainsi, au docker build, une copie statique de tout le contenu du dossier local *src/* est faite dans le dossier interne au container (*/var/www/html* ici).

Lancement de l'image

docker run -d -p 9090 :80 php:7.0-apache permet de lancer un container de cette image. En se connectant ensuite sur ce container, nous pouvons ensuite lui envoyer des requêtes. Pour ce faire, il faut récupérer l'adresse IP de la VM et faire la commande suivante : **telnet <ipVM> 9090**. Ensuite, il est possible d'envoyer une requête GET par exemple :

```
$ telnet 192.168.99.100 9090
Trying 192.168.99.100...
Connected to 192.168.99.100.
Escape character is '^]'.
GET / HTTP/.0
HTTP/1.1 400 Bad Request
Date: Sun, 04 Jun 2017 12:44:59 GMT
Server: Apache/2.4.10 (Debian)
Content-Length: 302
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
</p>
<hr>
<address>Apache/2.4.10 (Debian) Server at 172.17.0.2 Port 80</address>
</body></html>
Connection closed by foreign host.
```

Figure 2 Résultat telnet

Comme il n'a aucun contenu, le serveur retourne un message d'erreur.

Fichiers de configuration

Si l'on souhaite aller voir le contenu des fichiers de configuration il faut se connecter sur le container en exécution. Tout d'abord, pour connaître le nom du container, il faut faire un **docker ps**. Puis, en connaissant ce nom, il faut utiliser la commande **docker exec -it <nomContainer> /bin/bash**. Cela démarre un terminal dans le container. Il suffit ensuite de faire un **ls** pour voir ce qui se trouve dedans. Au début, il n'y a rien. On peut créer un fichier .html et écrire dedans, cela s'affichera dans le browser sur l'adresse *<ipVM> :9090*.

Au redémarrage de ce container, toutes les modifications faites à l'intérieur de ce dernier seront perdues. Il faut donc garder une copie en local. C'est pourquoi, dans le Dockerfile, la ligne commençant par COPY est très importante (cf. ci-dessus).

Afin de trouver les fichiers de configuration de l'image, il faut, une fois le bash démarré sur le container, utiliser la commande **cd** pour se déplacer dans les dossiers.

Dans le dossier /etc/apache2/ du container, il y a un fichier de configuration principal. De plus, dans le dossier *sites-available* il y a les sous-dossiers de configuration. De base, il n'y en a qu'un seul, *000-default.conf* qui contient le host virtuel par défaut. Cela permet de faire de traduire afin de savoir, lorsque l'on utilise une adresse URL, sur quelle adresse ip il faut rediriger la requête de demande d'accès.

Ce qu'il faut surtout noter, c'est que toute la configuration est dans le dossier /etc/apache2.

Notre configuration

```
$ docker exec -it unruffled_pike /bin/bash
time="2017-06-04T15:27:48+02:00" level=info msg="Unable to use system certificate pool: cryp
9: system root pool is not available on Windows"
root@8877baf72932:/var/www/html# ls
LICENSE  README.md  css  gulpfile.js  img  index.html  js  less  package.json  sass  vendor
root@8877baf72932:/var/www/html# cd etc/
bash: cd: etc/: No such file or directory
root@8877baf72932:/var/www/html# cd /etc/apache2
root@8877baf72932:/etc/apache2# ls
apache2.conf    conf-enabled  magic          mods-enabled  sites-available
conf-available  envvars       mods-available  ports.conf    sites-enabled
root@8877baf72932:/etc/apache2# cd sites-available
root@8877baf72932:/etc/apache2/sites-available# ls
000-default.conf  default-ssl.conf
root@8877baf72932:/etc/apache2/sites-available#
```

Figure 3 Commandes pour trouver les configurations

```
<VirtualHost *:80>
        # The ServerName directive sets the request scheme, hostname and port that
        # the server uses to identify itself. This is used when creating
        # redirection URLs. In the context of virtual hosts, the ServerName
        # specifies what hostname must appear in the request's Host: header to
        # match this virtual host. For the default virtual host (this file) this
        # value is not decisive as it is used as a last resort host regardless.
        # However, you must set it for any further virtual host explicitly.
        #ServerName www.example.com

        ServerAdmin webmaster@localhost
        DocumentRoot /var/www/html

        # Available loglevels: trace8, ..., trace1, debug, info, notice, warn,
        # error, crit, alert, emerg.
        # It is also possible to configure the loglevel for particular
        # modules, e.g.
        #LogLevel info ssl:warn

        ErrorLog ${APACHE_LOG_DIR}/error.log
        CustomLog ${APACHE_LOG_DIR}/access.log combined

        # For most configuration files from conf-available/, which are
        # enabled or disabled at a global level, it is possible to
        # include a line for only one particular virtual host. For example the
        # following line enables the CGI configuration for this host only
        # after it has been globally disabled with "a2disconf".
        #Include conf-available/serve-cgi-bin.conf
</VirtualHost>

# vim: syntax=apache ts=4 sw=4 sts=4 sr noet
```

Figure 4 Contenu du fichier 000-default.conf

Comme vous pouvez le voir sur les deux images précédentes, il n'est pas très compliqué de trouver et ouvrir les fichiers de configuration.

Le fichier par défaut est, comme susmentionné, le fichier *000-default.conf*. Vous pouvez lire son contenu sur l'image précédente.

Affichage html

Afin de créer une page html statique, nous avons, en local, créé un dossier nommé *src* dans lequel nous avons écrit un fichier nommé *index.html* dans lequel nous avons écrit du code basique HTML, afin d'avoir quelque chose qui s'affiche. Pour vérifier cet affichage, il faut build et run le container contenant le serveur apache et dont nous avons créé le fichier Dockerfile.

Il ne faut cependant pas oublier de mapper les ports en utilisant le flag **-p 9090 :80**. Si tout s'est bien passé, il suffit ensuite, via un browser, aller sur l'adresse ip du container (<ip> :9090). Le contenu du fichier index.html s'affichera ainsi dans la fenêtre.

Le serveur apache, dans docker, nous annoncera quant à lui toutes les requêtes qu'il aura reçues lors de la connexion, telles que les GET :

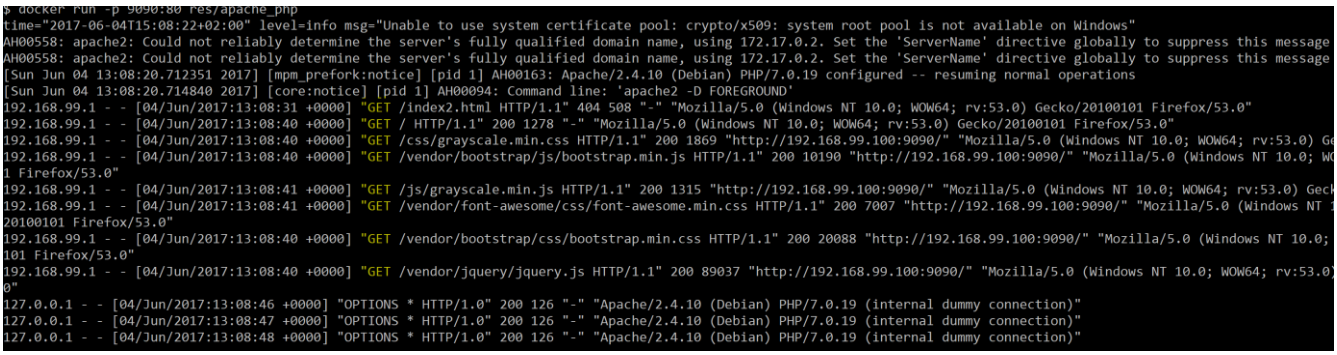

Figure 5 Résultat requêtes Firefox

Dans cette image, nous pouvons voir entre autres le navigateur qui a fait la requête (Firefox) ainsi que le code de retour annonçant le résultat de la requête. Dans notre cas, le code vaut 200, ce qui signifie que tout s'est bien passé.

Afin d'avoir une page HTML jolie et déjà créé, il est possible d'utiliser des templates préexistants et de les modifier afin d'avoir l'affichage et le contenu souhaité. Sur le conseil du professeur, nous avons utilisé un template issu de Bootstrap. Il nous a ensuite fallu uniquement enregistrer les divers dossiers fournis dans notre dossier local *src/*. Les changements ne sont cependant visibles uniquement après le rebuild et redémarrage du container.

Etape 2 : serveur http dynamique avec express.js

Branche GIT

Pour cette deuxième étape, vous pouvez trouver le résultat final sur la branche **fb-express-dynamic**.

Outils

Pour cette partie, nous avons utilisé **express.js**, qui est un framework qui fournit énormément de fonctionnalités pour les applications web.

Infrastructure générale

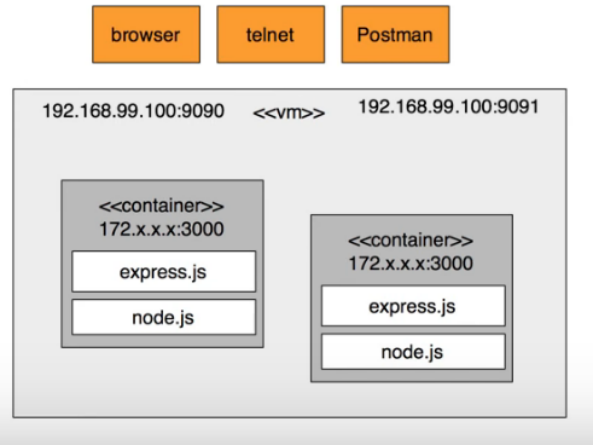

Figure 6 Structure générale étape 2

Dockerfile

Dans le dossier docker-images local, nous avons créé un nouveau dossier, *express-image*, dans lequel nous avons aussi fourni un fichier Dockerfile.

Voici son contenu :

```
FROM node:4.4
COPY src /opt/app
CMD ["node", "/opt/app/index.js"]
```

La première instruction, le FROM, annonce que nous utilisons la version 4.4 de node. De plus, nous copier ce qui est dans le dossier local *src* dans le dossier du container */opt/app*.

Afin de demander une exécution automatique du script que nous avons créé, nous avons aussi mis la ligne **CMD** dans le Dockerfile.

Configuration locale

Afin de créer un fichier **package.json** contenant les information nécessaires à l'utilisation des modules npm nous avons utilisé en local dans le dossier src la commande **npm init**. Ensuite, comme nous savions que nous aurions besoin du module C*hance* il nous a suffi de lancer la commande **npm install –save chance**, ce qui a eu pour conséquences de créer un dossier contenant les modules nécessaires ainsi que la complétion du package.json.

Voici le contenu final de ce fichier :

```
{
  "name": "students",
  "version": "0.1.0",
  "description": "Demo",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Rémi Jacquemard",
  "license": "ISC",
  "dependencies": {
    "chance": "^1.0.8",
    "express": "^4.15.3"
  }
```

```
}
```

Nous avons aussi créé un fichier, **index.js**, qui est le script qui est lancé et qui contient notre application node.

Pour tester le fonctionnement de cette première partie, il nous a fallu build et run l'image docker après avoir fait un petit code dans le fichier index.js. Ainsi, au lancement de l'image, le script affiche un message et nous avons pu ainsi valider cette partie.

De plus, en ayant lancé le container via une commande exec et un bash, nous pouvons retrouver les fichiers index.js, package.json, etc.

Application Node

Notre application Node a été écrite dans le fichier nommé **index.js**. Voici son contenu :

```
var Chance = require('chance');
var chance = new Chance();
var express = require('express');
var app = express();

app.get('/', function (req, res) {
   res.send(getHackedComputers());
});

app.listen(3000, function () {
   console.log("Accepting HTTP requests on port 3000");
});

function getHackedComputers() {
   var nbOfComputer = chance.integer({
      min: 1, max: 10
   });
   var computers = [];
   for (var s = 0; s < nbOfComputer; s++) {
      var nbWebSite = chance.integer({
         min: 1,
         max: 5
      });
      var webSites = [];
      for (var i = 0; i < nbWebSite; i++) {
         webSites.push(chance.url());
      }
      var computer = {
         'ip': chance.ip(),
         'localisation': {
         'lat': chance.latitude(),
         'lon': chance.longitude()
         },
         'lastWebSitesVisited': webSites,
         'lastPasswordUsed': chance.word() + chance.word()
      };
      computers.push(computer);
   }
   return computers;
}
```

Remarques

- Comme vous pouvez le voir, nous avons utilisé le module express. Il a donc fallu l'installer via la commande **npm install** dont nous avons déjà discuté précédemment.

- Les quatre premières lignes du code permettent de charger et de définir les modules utilisés dans le code. Ici il s'agit du module Chance et du module Express.
- *app.get* prend en premier paramètre la ressource visée par la requête GET et en deuxième paramètre une fonction de callback nomme *getHackedComputer* qui est définie plus bas dans le code. De cette manière, lorsqu'un GET vise la ressource représentée (ici, '/'), la fonction de callback est appelée.
- *app.listen* prend en premier paramètre le port sur lequel l'application se met à l'écoute. Le deuxième correspond à une fonction de callback créant un affichage log.
- La fonction *getHackedComputers* utilise le module Chance afin de générer des données aléatoires. Par exemple, il génère une adresse ip, des coordonnées, des url et des mots de passe aléatoires. Toutes ces données sont stockées dans un tableau.
  Cela permet d'obtenir à chaque fois des informations différentes et aléatoires, lesquelles sont, à la fin de cette étape, utilisées pour générer un JSON payload envoyé au client se connectant au serveur.

Le résultat final est, après le démarrage du serveur et la « connexion » depuis le browser sur ce dernier, un affichage de contenu dynamique.

Etape 3 : reverse proxy avec Apache (configuration statique)

Branche GIT

Pour cette troisième étape, vous pouvez trouver le résultat final sur la branche **fb-apache-reverse-proxy**.

Outils

Dans cette étape-ci, nous avons utilisé les requêtes AJAX. Ces dernières permettent d'envoyer des requêtes asynchrones au serveur et ainsi mettre à jour des parties uniquement de notre « site » html.

Il faut, pour pouvoir l'utiliser, passer par un reverse proxy.

Reverse proxy

Un reverse proxy permet de cacher l'infrastructure de l'extérieur. Ainsi, seul le reverse proxy est visible et c'est lui qui, en tant que point d'entrée unique, va rediriger les requêtes aux bons containers (en ce qui nous concerne).

Infrastructure générale

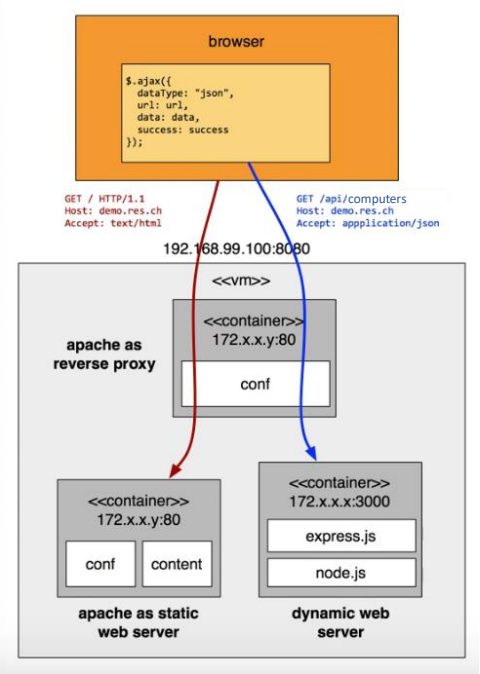

Figure 7 Structure générale étape 3

Comme vous pouvez le voir sur l'image, notre infrastructure présente un unique point d'entrée, qui est un reverse proxy. Les requêtes passent donc obligatoirement par ce point, et c'est ce dernier qui les redirige.

Configuration fragile et adresses IP

La configuration de cette étape est fragile. En effet, afin que tout fonctionne correctement, il faut respecter un ordre précis de démarrage des containers docker. Cela est dû au fait que nous avons hardcodé dans le fichier[1] de configuration gérant les VirtualHost les adresses IP alors qu'elles sont fournies dynamiquement par docker. Ainsi, il suffit qu'au démarrage d'un des containers son adresse IP ne corresponde plus à ce qui était prévu pour que plus rien ne fonctionne.

Ce problème sera réglé dans les étapes suivantes. Cela permettra de ne plus avoir de problème au niveau des adresses IP.

Afin d'obtenir les adresses IP d'un container en fonctionnement, il faut utiliser la commande **docker inspect <nomContainer> | grep -i ipaddress**.

Ce sont ces adresses qui ont un grand risque de changer à chaque démarrage des containers.

Sites

Lorsque nous sommes dans les fichiers du container Apache, comme nous l'avons mentionné plus tôt, nous pouvons trouver les dossiers de configuration dans *etc/apache2*. Dans ce dossier, il y a les sous-dossiers concernant les sites (pour les distributions ubuntu/debian).

Un serveur apache peut servir plusieurs sites logiques en même temps. Ainsi, dans le dossier site-enabled nous pouvons trouver les configurations pour les sites.

[1] 001-reverse-proxy.conf

Lorsque le client envoie une requête GET avec comme en-tête HOST une url, c'est les sites qui annoncent quelle redirection faire et quoi afficher, donc quel VirtualHost utiliser.

Afin de pouvoir avoir notre propre VirtualHost, nous avons dû créer un fichier nommé 001-reverse-proxy.conf, lequel est stocké en local dans un dossier nommé *conf*. Nous l'avons stocké en local, car ainsi il continue d'exister même en cas de redémarrage du serveur.

001-reverse-proxy.conf

Voici le contenu de notre nouveau VirtualHost (fichier 001-reverse-proxy.conf) :

```
<VirtualHost *:80>
   ServerName demo.res.ch

   ProxyPass "/api/computers/" "http://172.17.0.3:3000/"
   ProxyPassReverse "/api/computers/" "http://172.17.0.3:3000/"

   ProxyPass "/" "http://172.17.0.2:80/"
   ProxyPassReverse "/" "http://172.17.0.2:80/"
</VirtualHost>
```

- **ServerName** permet d'annoncer quelle URL est attendue pour faire les redirections
- **ProxyPass** dit que si l'URL commence par ce qui est entre guillemets alors il faut se rediriger à l'adresse IP et le port entre guillemets aussi.
- **ProxyPassReverse** permet de dire quoi faire lorsqu'une réponse doit être envoyé. C'est le même principe que pour ProxyPass.
- Attention, les règles doivent être dans le bon ordre : le plus restrictif en premier, le moins en dernier.

Une fois ce fichier créé, il faut activer le site via la commande **a2ensite 001***. a2ensite permet d'activer des sites logiques, lesquels sont servis par le serveur Apache. Ensuite, il faut activer les modules nécessaires : proxy et proxy_http en utilisant la commande **a2enmod <module>**. Pour finir, il faut utiliser la commande **service apache2 reload**. La configuration est maintenant terminée et fonctionnelle. Elle est testable via un telnet sur l'adresse ip de la VM. Cela nous connecte au reverse proxy, et lorsque nous faisons une requête GET fonctionnelle, il nous redirige au bon composant et nous recevons une réponse correspondante.

000-default.conf

Nous avons aussi gardé le fichier 000-default.conf. Voici son contenu :

```
<VirtualHost *:80>
</VirtualHost>
```

En effet, sans ce fichier, le VirtualHost par défaut serait un de ceux présent dans le fichier 001-reverse-proxy.conf. Cela aurait pour conséquence que si le client n'utilisait pas l'en-tête host demo.res.ch dans sa requête, il arriverait quand même dans la configuration de 001. Afin de protéger et rendre plus strict l'accès au serveur, il vaut donc mieux blinder et garder ce fichier de VirtualHost par défaut.

Traduction d'adresses IP

Sur Windows 10, en local, dans le dossier *C:\Windows\System32\drivers\etc* vous pouvez trouver un fichier nommé *hosts*. Ce fichier permet de traduire les urls avec les IPs correspondantes.

Nous avons ajouté dans ce fichier la ligne suivante :

```
192.168.99.100 demo.res.ch
```

Cela permet de dire que si on donne l'url demo.res.ch il doit être traduit avec l'adresse IP 192.168.99.100. Il s'agit en fait de l'adresse IP de la machine virtuelle de docker.

Dockerfile

Les fichiers de cette étape sont contenus dans le dossier local nommé *apache-reverse-proxy.* C'est d'ailleurs dans ce dossier que nous avons créé le fichier Dockerfile correspondant au container que nous avons créé dans cette étape.

Voici le contenu du fichier Dockerfile :

```
FROM php:5.6-apache

COPY conf/ /etc/apache2

RUN a2enmod proxy proxy_http
RUN a2ensite 000-* 001-*
```

Comme pour les étapes précédentes, nous avons les deux mêmes premières lignes (FROM et COPY). Nous avons aussi deux lignes commençant par RUN. Cela permet de lancer les deux commandes susmentionnées, à savoir l'activation de module (a2enmod) et l'activation de sites logiques (a2ensite). De cette manière, ces commandes sont effectuées automatiquement au démarrage de l'image docker.

Lancement

Une fois l'image build, il suffit ensuite de lancer le container sans oublier de mapper les ports. Ensuite, via un browser, il est possible d'afficher le site que nous avons créé.

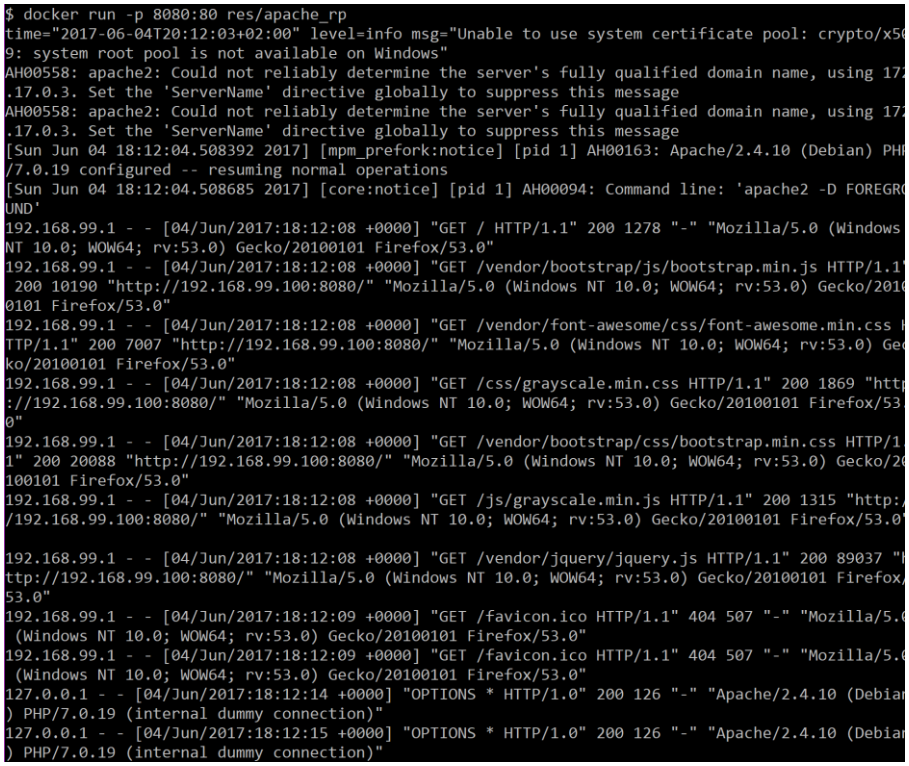

Figure 8 Réception des requêtes du browser Etape 3

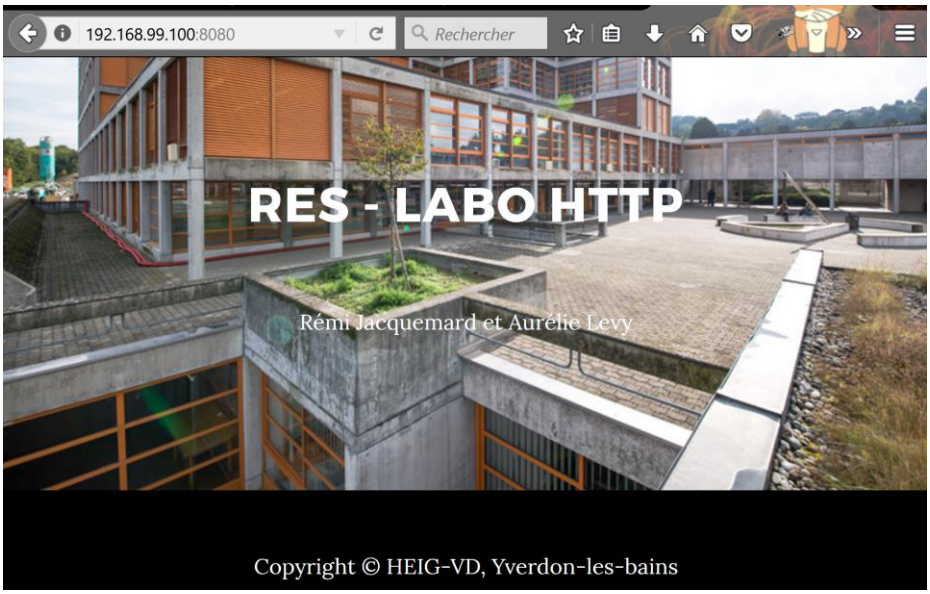

Figure 9 Visuel du site Etape 3

~~ESSAYER DE FAIRE CA (13 minutes 3c) :~~

```
$ telnet 192.168.99.100 8080
Trying 192.168.99.100...
Connected to 192.168.99.100.
Escape character is '^]'.
GET /api/computers/ HTTP/1.0
Host: demo.res.ch
Connection closed by foreign host.
```

Etape 4 : AJAX avec JQuery

Branche GIT

Pour cette quatrième étape, vous pouvez trouver le résultat final sur la branche **fb-ajax-jquery**.

Outils

Pour cette étape, nous avons utilisé JQuery afin de pouvoir faire des requêtes AJAX et mettre ainsi à jour une partie de la fenêtre HTML (élément DOM) affichée par le browser.

Restriction de sécurité

Selon la *Same Origin Policy*, les requêtes *Ajax* doivent être exécutée sur le même serveur que la page *HTML*. Ainsi, l'utilisation d'un reverse proxy afin de faire paraitre les deux serveurs (*apache_php* et *express_dynamic*) comme étant un seul permet de respecter cette politique de sécurité.

Dockerfile

Modifications

Dans l'image apache-php-image, nous avons dans cette étape mis une commande (RUN) supplémentaire pour pouvoir utiliser vim :

```
FROM php:7.0-apache

RUN apt-get update && \
   apt-get install -y vim

COPY src/ /var/www/html/
```

Idem pour le Dockerfile de l'image apache-reverse-proxy et pour celui de express-image.

Marche à suivre

Pour commencer cette étape, il faut d'abord démarrer les containers dans le bon ordre (toujours à cause des adresses IP hardcodées) et de vérifier le fonctionnement du tout. En effet, les adresses IP peuvent avoir changé, c'est pourquoi il faut être prudent à ce niveau-là.

~~Mettre des printscreens des commandes docker et du resultat sur le browser~~

Une fois les trois containers démarrés, nous avons pu commencer à faire l'étape réellement.

Dans le fichier index.html utilisé par le serveur apache, nous pouvons trouver des lignes contenant des balises <script></script>. Ces dernières permettent d'inclure des scripts propres à l'application.

Nous avons donc ajouté la ligne suivante au fichier HTML :

```
    <!-- Custom script to load computers -->
    <script src="js/computers.js"></script>
```

Ensuite dans le sous-dossier de *src*, nommé *js*, nous avons créé computers.js. Voici son contenu :

```
$(function() {
    console.log("Loading computers");

    function loadComputers(){
        $.getJSON("/api/computers/", function(computers){
            console.log(computers);

            var message = "No computer hacked";
            if(computers.length > 0){
                message = computers[0].ip + " used the password : '" +
computers[0].lastPasswordUsed + "'";
            }

            $(".intro-text").text(message);
        });
    };

    loadComputers();

    setInterval(loadComputers, 2000);
});
```

- **$(function(){** : cela signifie que lorsque JQuery a été chargé ($) il faut exécuter la fonction passée en callback.
- Cette fonction callback consiste principalement au lancement de la fonction nommée loadComputers toutes les 2000 ms (**setInterval**).
- **loadComputers** utilise **$.getJSON** avec en paramètre une URL et une fonction de callback. Ainsi, le moteur javascript va déclencher l'envoi/le chargement de la requête http de manière asynchrone, et quand les données sont de retour elles sont passées à la fonction de callback passée en paramètre.
- **$(« .intro-text »(.text(message) ;** fait en sorte de remplacer le texte présent dans la balise HTML possédant la classe *intro-text* par le texte contenu dans la variable *message*. Cette variable est modifiée par la fonction elle-même et contient, normalement, une ip, un texte « used the password : » et un mot de passe d'un ordinateur aléatoire, obtenu par les requêtes AJAX précédemment citées.

Etape 5 : configuration du reverse proxy dynamique

Branche GIT

Pour cette cinquième étape, vous pouvez trouver le résultat final sur la branche **fb-ajax-jquery**.

~~Infrastructure générale~~

~~Faire un schéma~~

Configuration

Adresses IP dynamiques

L'objectif de l'étape est de remplacer les IP hardcodées et de les rendre dynamiques.

Tout d'abord, il faut savoir qu'en démarrant un container, il est possible via le flag **-e** de passer en paramètre des variables d'environnement.

```
$ docker run -e Hello=world -it res/apache_rp /bin/bash
time="2017-06-04T21:23:35+02:00" level=info msg="Unable to use system certificate pool: crypto/x5
9: system root pool is not available on Windows"
root@e570b36b4d75:/var/www/html# export
declare -x APACHE_CONFDIR="/etc/apache2"
declare -x APACHE_ENVVARS="/etc/apache2/envvars"
declare -x GPG_KEYS="1A4E8B7277C42E53DBA9C7B9BCAA30EA9C0D5763 6E4F6AB321FDC07F2C332E3AC2BF0BC433C
C8B3"
declare -x HOME="/root"
declare -x HOSTNAME="e570b36b4d75"
declare -x Hello="world"
declare -x OLDPWD
declare -x PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
declare -x PHPIZE_DEPS="autoconf                dpkg-dev                file            g++
gcc             libc-dev                libpcre3-dev            make            pkg-config
re2c"
declare -x PHP_ASC_URL="https://secure.php.net/get/php-7.0.19.tar.xz.asc/from/this/mirror"
declare -x PHP_CFLAGS="-fstack-protector-strong -fpic -fpie -O2"
declare -x PHP_CPPFLAGS="-fstack-protector-strong -fpic -fpie -O2"
declare -x PHP_EXTRA_BUILD_DEPS="apache2-dev"
declare -x PHP_EXTRA_CONFIGURE_ARGS="--with-apxs2"
declare -x PHP_INI_DIR="/usr/local/etc/php"
declare -x PHP_LDFLAGS="-Wl,-O1 -Wl,--hash-style=both -pie"
declare -x PHP_MD5="1a17e45c8be9ce28f036d884563e8ae7"
declare -x PHP_SHA256="640e5e3377d15a6d19adce2b94a9d876eeddabdb862d154a5e347987f4225ef6"
declare -x PHP_URL="https://secure.php.net/get/php-7.0.19.tar.xz/from/this/mirror"
declare -x PHP_VERSION="7.0.19"
declare -x PWD="/var/www/html"
declare -x SHLVL="1"
declare -x TERM="xterm"
root@e570b36b4d75:/var/www/html#
```

Figure 10 Exemple de l'utilisation du flag -e

Nous pouvons utiliser ces variables d'environnement afin de passer au container reverse proxy les adresses IP des deux containers :

- res/apache_php : -e STATIC_APP=172.17.0.2
- res/express_computers : -e DYNAMIC_APP=172.17.0.3

~~Lorsque l'on va regarder le repo de l'image que nous utilisons pour le serveur Apache (version php :5.6), on peut voirque dans le Dockerfile que les auteurs ont utilisé ils démarrent Apache en avant-plan (**CMD [« aache2 -foreground »]**) afin que le container ne termine pas tout de suite. Nous avons donc utilisé~~

On se souvient que la configuration du reverse proxy s'effectuait localement dans le fichier *conf/sites-avaible/001-reverse-proxy.conf* : c'est dans ce fichier qu'on trouve les IPs des containers dynamiques et statiques.

Afin d'utiliser les variables d'environnement plutôt que des adresses IPs codées en dur, PHP a été utilisé. Un fichier, dans le dossier *templates/config-template.php* a été créé, contenant la même configuration qu'auparavant, même obtenant les IPs dynamiquement. En voici le contenu :

```
<?php
    $static_app = getenv('STATIC_APP');
    $dynamic_app = getenv('DYNAMIC_APP');
?>

<VirtualHost *:80>
    ServerName demo.res.ch

    ProxyPass "/api/computers/" "http://<?php print $dynamic_app ?>:3000/"
    ProxyPassReverse "/api/computers/" "http://<?php print $dynamic_app
?>:3000/"


    ProxyPass "/" "http://<?php print $static_app ?>:80/"
    ProxyPassReverse "/" "http://<?php print $static_app ?>:80/"
 </VirtualHost>
```

Le dossier *templates* peut maintenant être copié dans le container, dans le dossier */var/apache2/templates*. Voici la ligne ajoutée au *Dockerfile* :

```
COPY templates /var/apache2/templates
```

Le container *res/apache_rp* utilise, comme décrit dans son *Dockerfile*, l'image *php:5.6-apache*. Afin d'utiliser les variables d'environnement pour les IPs des deux containers, on peut explorer le *Dockerfile* associé et les fichiers afin de savoir quoi copier, et où. L'image de *php* est disponible sur *github* :

https://github.com/docker-library/php/tree/master/5.6/apache

On remarque, en fin du *Dockerfile*, que la commande *apache2-foreground* est lancée. C'est en fait un fichier, qu'il est possible de remplacer à la construction du container.

Afin d'avoir un fichier de configuration reverse proxy créé dynamiquement, il est maintenant nécessaire d'exécuter le script PHP défini auparavant, et que la sortie de ce script soit dans un fichier de configuration *.conf*, dans le dossier */etc/apache2/sites-avaible*, comme demandé par *apache*. On peut y remplacer notre ancien fichier *001-reverse-proxy.conf*. Nous avons donc copié le fichier original dans le dossier *apache-reverse-proxy*, et pu y ajouter la ligne suivante :

```
php /var/apache2/templates/config-template.php > /etc/apache2/sites-
available/001-reverse-proxy.conf
```

Il est maintenant important de copier ce fichier modifié à la place de l'ancien. Pour se faire, on peut utiliser la commande COPY dans le *Dockerfile* de notre image :

```
COPY apache2-foreground /usr/local/bin/
```

Etape 6 : Load balancing : ajout de serveurs multiples

Pour cette étape il était demandé d'avoir un système de répartition de charge. Pour ce faire, on a doublé le nombre de serveur :

- 2 serveurs statiques *apache_php* : IP 172.17.0.2 et 172.17.0.3
- 2 serveurs dynamiques *express_computers* : 172.17.0.4 et 172.17.0.5

Afin d'activer la répartition de charge, le module *proxy_balancer* a été ajouté au *Dockerfile* du reverse proxy :

```
RUN a2enmod proxy_balancer
```

Afin de faire fonctionner cette étape, nous avons dû faire divers modifications/ajouts à notre fichier *config-template*. Les voici :

1. Modification du fichier config-template, avec ajout des *balancers* et des variables d'environnements.
2. Modification des *Proxy Pass* pour prendre en compte les *balancers* précédemment définis.

```
<?php
    $static_app_1 = getenv('STATIC_APP_1');
    $static_app_2 = getenv('STATIC_APP_2');
    $dynamic_app_1 = getenv('DYNAMIC_APP_1');
    $dynamic_app_2 = getenv('DYNAMIC_APP_2');
?>

<VirtualHost *:80>

    <Proxy "balancer://static">
        BalancerMember "http://<?php print $static_app_1 ?>:80"
        BalancerMember "http://<?php print $static_app_2 ?>:80"
        ProxySet lbmethod=bytraffic
    </Proxy>

    <Proxy "balancer://dynamic">
        BalancerMember "http://<?php print $dynamic_app_1 ?>:3000"
        BalancerMember "http://<?php print $dynamic_app_2 ?>:3000"
        ProxySet lbmethod=bytraffic
    </Proxy>

    ServerName demo.res.ch
```

```
    ProxyPass "/api/computers/" "balancer://dynamic/"
    ProxyPassReverse "/api/computers/" "balancer://dynamic/"

    ProxyPass "/" "balancer://static/"
    ProxyPassReverse "/" "balancer://static/"
</VirtualHost>
```

Afin de lancer le reverse proxy avec les bons paramètres, utiliser la commande suivante :

- docker run -e STATIC_APP_1=172.17.0.2 -e STATIC_APP_2=172.17.0.3 -e DYNAMIC_APP_1=172.17.0.4 -e DYNAMIC_APP_2=172.17.0.5 -p 8080:80 res/apache_rp

Etape 7 : Load balancing : round-robin vs sticky-session

L'implémentation courante du load balancing n'est pas parfaite. En effet, notre page web pourrait avoir un état (par exemple, un panier d'achat). Ainsi, l'utilisation de sticky-session est dans ce cas appropriée:

- Container *apache_php* : utilisation de sticky-session
- Container *express_computers* : utilisation de round-robin, pas d'importance entre les sessions car il n'y a pas d'état.

Nous avons dû modifier le fichier *config-template* pour prendre en compte les sticky-sessions :

- Ajout d'un header qui stockera l'id de la route à prendre
- Ajout de numéro de route au 2 route des containers apache_php
- Colle la sticky-session définie par ROUTEID au proxy pass des containers statiques

```
<?php
    $static_app_1 = getenv('STATIC_APP_1');
    $static_app_2 = getenv('STATIC_APP_2');
    $dynamic_app_1 = getenv('DYNAMIC_APP_1');
    $dynamic_app_2 = getenv('DYNAMIC_APP_2');
?>

<VirtualHost *:80>
    Header add Set-Cookie "ROUTEID=.%{BALANCER_WORKER_ROUTE}e; path=/"
env=BALANCER_ROUTE_CHANGED

    <Proxy "balancer://static">
        BalancerMember "http://<?php print $static_app_1 ?>:80" route=1
        BalancerMember "http://<?php print $static_app_2 ?>:80" route=2
        ProxySet lbmethod=bytraffic
    </Proxy>

    <Proxy "balancer://dynamic">
        BalancerMember "http://<?php print $dynamic_app_1 ?>:3000"
        BalancerMember "http://<?php print $dynamic_app_2 ?>:3000"
        ProxySet lbmethod=bytraffic
    </Proxy>

    ServerName demo.res.ch
```

```
    ProxyPass "/api/computers/" "balancer://dynamic/"
    ProxyPassReverse "/api/computers/" "balancer://dynamic/"

    ProxyPass "/" "balancer://static/" stickysession=ROUTEID
    ProxyPassReverse "/" "balancer://static/"
</VirtualHost>
```

Note : plutôt que de créer des numéros de route dans les *BalencerMember*, les containers *apache_php* auraient pu eux-mêmes envoyer un id de session, qui servirait au *reverse proxy* à qui envoyer les requêtes.

Il a aussi été nécessaire d'ajouter le module *header* dans le *Dockerfile* :

```
RUN a2enmod headers
```