

RES

Web Infrastructure Lab

Auteurs :

Simon Baehler

Armand Delessert

Raphaël Racine

Olivier Djeulezeck

Destinataires :

Olivier Liechti

Simon Oulevay

Laurent Prévost

|  |  |
| --- | --- |
| Du | mercredi 6 mai 2015 |
| Au | vendredi 27 mai 2015 |

# Table des matières

[Table des matières 1](#_Toc420889416)

[1. Objectifs 3](#_Toc420889417)

[2. Architecture 4](#_Toc420889418)

[3. Développement 5](#_Toc420889419)

[3.1. Back-end 5](#_Toc420889420)

[3.1.1. Structure des fichiers 5](#_Toc420889421)

[3.1.2. Dockerfile 5](#_Toc420889422)

[3.1.3. Start.sh 6](#_Toc420889423)

[3.1.4. Test 6](#_Toc420889424)

[3.1. Front-end 7](#_Toc420889425)

[3.1.1. Structure des fichiers 7](#_Toc420889426)

[3.1.2. Dockerfile 7](#_Toc420889427)

[3.1.3. Start.sh 8](#_Toc420889428)

[3.1.4. Test 9](#_Toc420889429)

[3.2. Reverse proxy 10](#_Toc420889430)

[3.2.1. Structure des fichiers 10](#_Toc420889431)

[3.2.2. Load Balancer 10](#_Toc420889432)

[3.2.3. Dockerfile 11](#_Toc420889433)

[3.2.4. Test 11](#_Toc420889434)

[3.3. Hearbeat front-end et back-end 12](#_Toc420889435)

[3.3.1. Heartbeats de NodeJS 12](#_Toc420889436)

[3.3.2. Heartbeat 12](#_Toc420889437)

[3.3.3. Test 13](#_Toc420889438)

[4. Conclusion 14](#_Toc420889439)

[4.1. Simon Baehler 14](#_Toc420889440)

[4.2. Raphaël Racine 14](#_Toc420889441)

[4.3. Armand Delessert 14](#_Toc420889442)

[4.4. Olivier Djeuzeck 14](#_Toc420889443)

# Objectifs

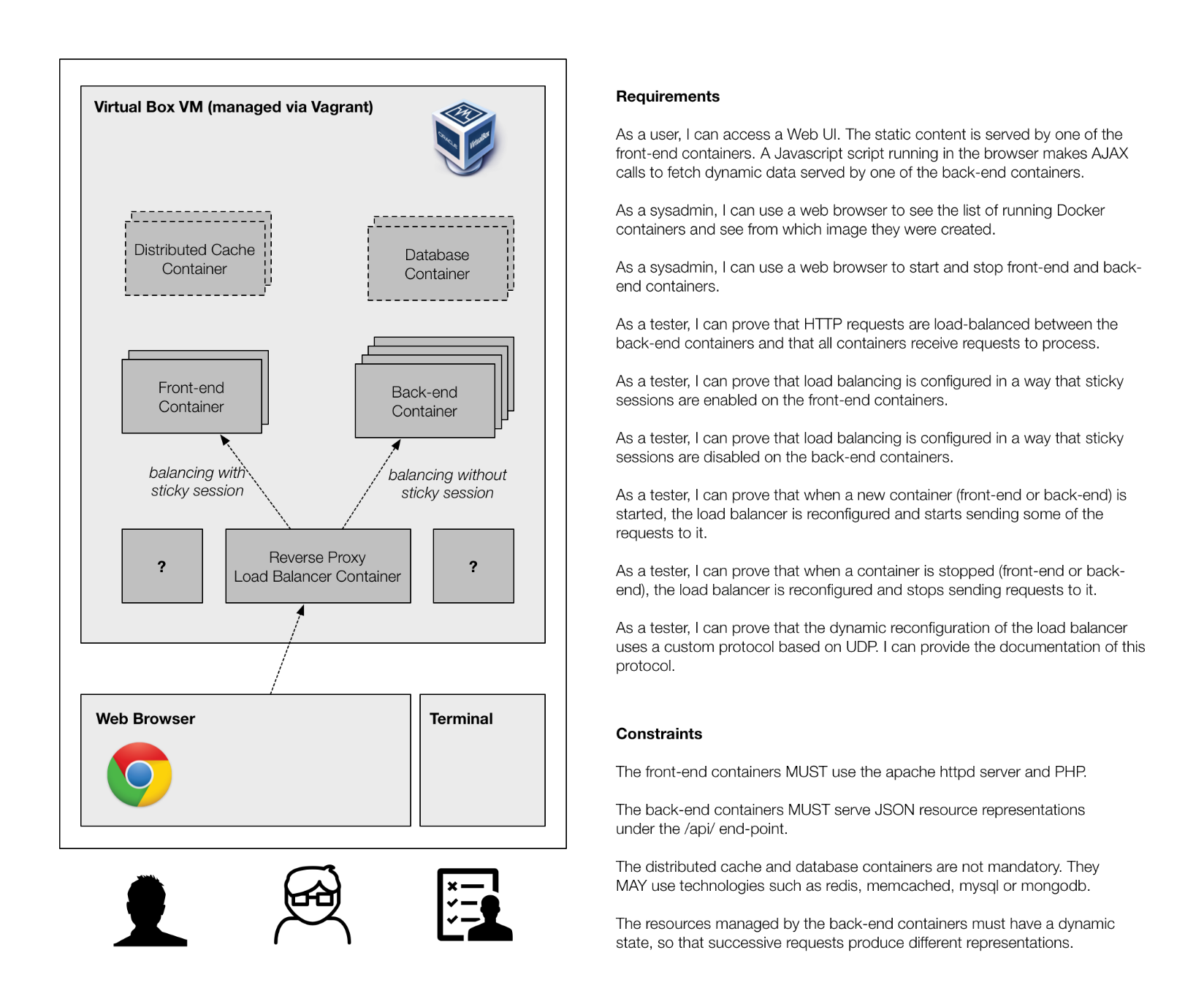
L’objectif principal de ce labo est d’expérimenter la mise en place d’une infrastructure web et de se familiariser avec les différents composants de celle-ci (serveurs http, proxys inverses, équilibreurs de charge).

Le second objectif est de se familiariser avec les outils Vagrant et Docker. En effet, l’infrastructure web à réaliser sera mise en place dans une machine virtuelle Vagrant et via des containers Docker pour chacun des composants de l’infrastructure web.

Enfin, le troisième objectif est d’implémenter la découverte dynamique de service via UDP. Il devra être possible, dans l’infrastrucutre web mise en place, d’ajouter à la volée des serveurs front end ou back end encapsulé dans des containers Docker.

# Architecture

L’architecture de l’infrastructure web à mettre en place est présentée dans l’image suivante.



Le client se connecte à notre service web en entrant l’URL (<http://exemple.com/>) dans son navigateur web. Le navigateur web va alors envoyer une requête GET au reverse proxy qui va la faire suivre au front end et logger l’adresse IP du navigateur web pour les futures connexions de ce dernier. Le front end retourne au navigateur web une page HTML contenant un script JavaScript qui va envoyer des requêtes au back end en s’adressant à l’URL <http://exemple.com/api>.

# Développement

## Back-end

Cette section concerne ce que nous avons réalisé pour la partie back-end.

Nous avons décidé dans un premier temps d’écrire un script en Javascript qui lorsqu’il reçoit une requête http sur le port 80 avec comme URI un ‘/’ , il répond avec un Json qui contient le nom d’un étudiant tiré au hasard parmi une liste d’étudiant.

### Structure des fichiers

Répertoire racine du back-end, nommé « back\_end »

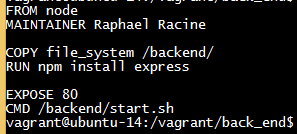


Contenu du répertoire file\_system



### Dockerfile

Voici le Dockerfile réalisé pour la partie back-end :



Premièrement, on fait un FROM node pour avoir l’outil node qui permet de lancer le script d’un fichier Javascript.

Ensuite, on copie tout le contenu du dossier file\_system dans un répertoire nommé backend, qui se trouvera donc dans l’image à partir de laquelle sera lancé le container.

Ensuite, on fait une installation de express de qui nous permettra de lancer le fichier javascript.

On fait un EXPOSE 80 pour dire que le container est exposé sur le port 80.

Pour finir, on fait un CMD /backend/start.sh pour que le container exécute cette commande lorsqu’on le lance, donc il exécutera le start.sh qui contient les informations de lancement des deux scripts Javascript.

### Start.sh

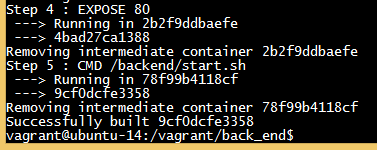
Les commandes qui se trouvent dans ce fichier shell sont lancées lorsque le container et démarré. Le contenu de ce fichier est :



On démarre l’outil node en lancant d’abord le script back\_end.js (qui se trouve dans le répertoire backend qui avait été copié dans l’image), et on fait la même chose pour le HB.js.

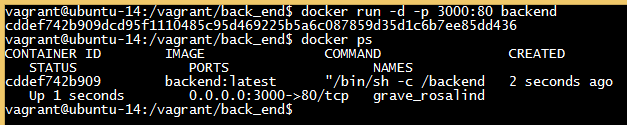
### Test

Tout d’abord, on construit une image nommé backend en tapant en ligne de commande docker build –t backend . :

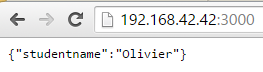


L’image a bien été créé.

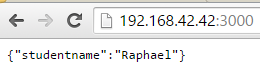
Ensuite, on lance notre back-end grâce à la commande   
docker run –d –p 3000:80 backend



On voit que le container a bien été lancé et qu’il écoute sur le port 80. Il a été mappé avec le port 3000 sur la machine Vagrant, on y a donc accès depuis la machine physique qui a lancé la machine vagrant en accèdent simplement à l’url l’IP 192.168.42.42:3000.



On voit que le backend a bien répondu en envoyant du contenu Json.



Après rafraîchissement de la page, on voit que le back-end a renvoyé un autre étudiant, tiré au hasard dans une liste.

Concernant la partie « heartbeat », consulter la section correspondante.

## Front-end

Cette section concerne ce que nous avons réalisé pour la partie front-end.

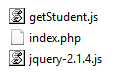
Nous avons décidé d’écrire une page php (index.php) qui possède un bouton sur lequel on cliquera dessus pour envoyer une requête AJAX à un backend (via le reverse proxy, voir plus loin) http avec l’URI ‘/api’ que le reverse proxy redirigera vers un des backend avec l’URI ‘/’, comme le backend traite les requêtes http qu’il reçoit sur l’URI ‘/’.

### Structure des fichiers

Répertoire racine du front-end, nommé « front\_end »

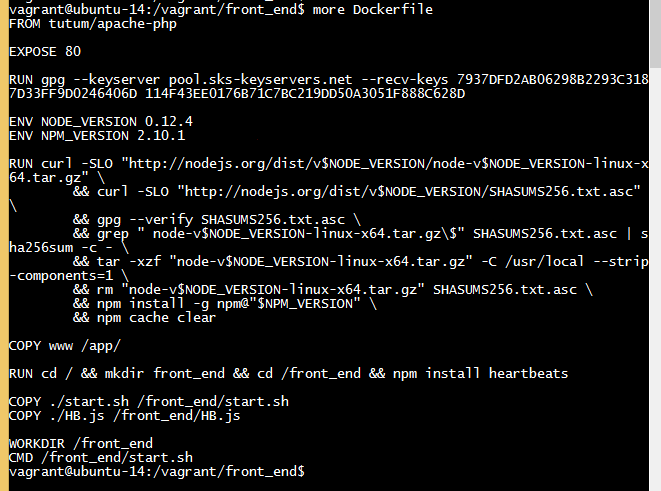


Contenu du répertoire www



### Dockerfile

Voici le Dockerfile réalisé pour la partie front-end :



Premièrement, on fait un FROM tutum/apache-php pour avoir à disposition un serveur Apache pour afficher la page web avec le module php (ce module est nécessaire pour afficher l’adresse IP du frontend qui répond)

Ensuite, on dit qu’un container lancé à partir de cette image sera exposé sur le port 80.

Toutes les instruction d’après (jusqu’au COPY) permettent d’installer nodejs pour lancer le script HB.js.

Ensuite, on copie le répertoire www dans /app/ du container, répertoire à partir duquel le serveur Apache enverra la page Web.

La commande RUN Cd … permet de nous placer à la racine, d’y créer un répertoire front\_end, de se déplacer dans ce dernier puis d’y installer heartbeats.

On copie ensuite les fichiers start.sh et HB.js dans un répertoire front\_end dans le container.

Tout comme le back-end, on fait ensuite un CMD /front\_end/start.sh pour lancer les instructions shell du fichier start.sh lors du démarrage du container.

### Start.sh

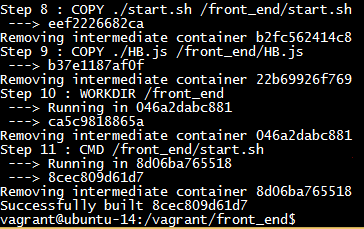
Les commandes qui se trouvent dans ce fichier shell sont lancées lorsque le container et démarré. Le contenu de ce fichier est :



On démarre d’abord le serveur apache et ensuite le script HB.js avec l’utilitaire node.

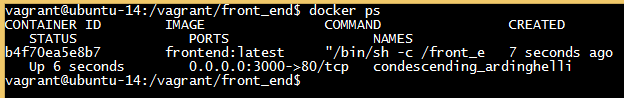
### Test

Tout d’abord, on construit une image nommé frontend en tapant en ligne de commande docker build –t frontend . :



L’image a bien été créée.

Ensuite, on lance notre front-end grâce à la commande   
docker run –d –p 3000:80 frontend



On voit que le container a bien été lancé et qu’il écoute sur le port 80. Il a été mappé avec le port 3000 sur la machine Vagrant, on y a donc accès depuis la machine physique qui a lancé la machine vagrant en accèdent simplement à l’url l’IP 192.168.42.42:3000.



On voit que le frontend a bien répondu en envoyant du contenu html.

## Reverse proxy

Cette section concerne ce que nous avons réalisé pour la partie reverse proxy.

Pour cette partie, nous avons décidé d’avoir un serveur Apache qui écoute sur le port 80 et dont le fichier de configuration **httpd.conf** sera monté avec un volume lors du lancement du container, ceci afin de pouvoir modifier ce fichier depuis le controlleur du heartbeat, qui se trouverait dans un autre container.

En effet, si on aurait copié de fichier dans l’image, on aurait pas pu le modifier depuis un autre container.

### Structure des fichiers

Voici les fichiers se trouve à la racine du reverse proxy :



Et enfin le contenu du dossier config :



### Load Balancer

Voici un « template » du fichier httpd.conf de la manière dont il devrait être configuré via le heartbeat  (seul la partie concernant le reverse-proxy est montrée ici)



Voir les explications dans « Test ».

### Dockerfile

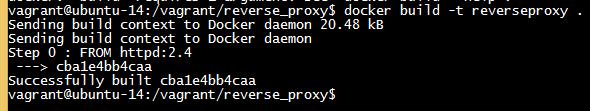
Voici le contenu du Dockerfile concernant la partie reverse proxy :



On se base simplement sur l’image Docker Hub nommé httpd :2.4, pour avoir un serveur Apache fonctionnel.

### Test

Tout d’abord, on construit une image nommé reverseproxy en tapant en ligne de commande docker build –t reverseproxy . :



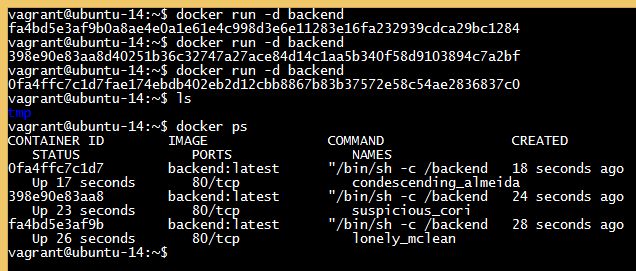
L’image a bien été créé.

Ensuite, comme nous n’avons malheureusement pas eu le temps de faire la partie contrôleur pour changer le fichier httpd.conf du reverse proxy dynamiquement lorsque des containers arrivent ou repartent, nous devons lancer d’abord les back-end, suivi des front-end, et enfin le reverseproxy que nous devons lancer à la fin en montant un volume vers le fichier httpd.conf après avoir introduit les IP des containers front-end et back-end (car Docker attribut à chaque fois une adresse IP différente de manière aléatoire).

En pratique, nous devons faire ce qui suit pour que ça fonctionne.

#### Démarrage des backend

On commence par créer par exemple 3 backend en lançant 3 fois la commande  
docker run –d backend

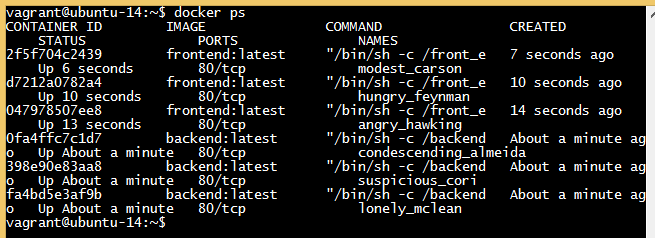


On a donc 3 backend qui écoutent sur le port 80.

#### Démarrage des frontend

On créer également par exemple 3 frontend en lançant 3 fois la commande

docker run –d frontend



On a donc 3 frontend qui écoutent sur le port 80.

#### Démarrage du reverse proxy

Comme nous devons modifier le fichier httpd.conf « à la main » du côté reverse proxy (car nous devons connaître les IP des containers lancés, nous lançons d’abord docker UI pour avoir connaissance de ces adresses.

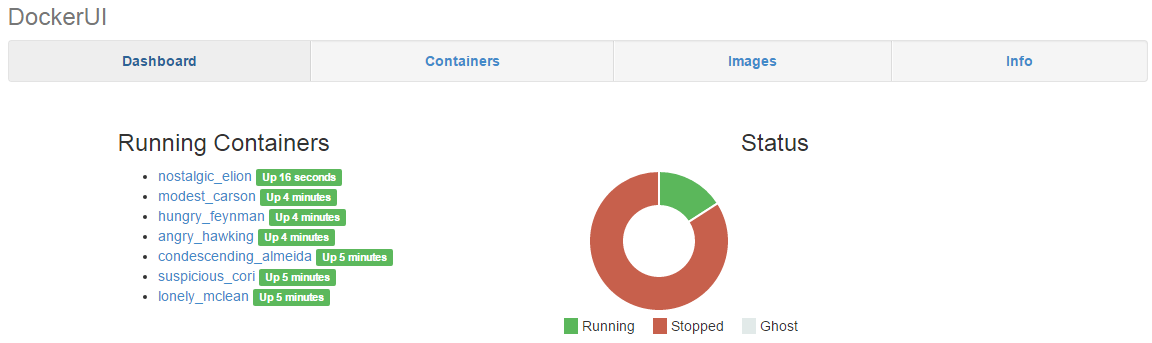
Cela nécessite de lancer la commande suivante :

docker run -d -p 9000:9000 --privileged -v /var/run/docker.sock:/var/run/docker.sock dockerui/dockerui

Cette commande créer un container qui lance un processus permettant d’aller sur Docker UI, qui est un outil pour visualiser les containers en cours d’exécution et bien d’autres choses.

On se connecte sur Docker UI dans un navigateur à l’aide de l’url suivante :

192.168.42.42:9000



On voit donc les containers en cours d’exécution, en cliquant sur un des front-end ou back-end (dans l’onglet containers), on peut voir leur adresse IP, un exemple pour front-end :



On voit que le front-end en question à l’adresse IP 172.17.0.6

Donc, on modifie notre fichier httpd.conf qui se trouve dans le reverse proxy dans le dossier reverse\_proxy/config et on configure la partie Load Balancer de la manière suivante :



Dans l’encadré **orange** on a mis une règle concernant le backend.

Cela signifie que chaque requête qui arrive sur ‘/api’ (selon le contenu de l’encadré **vert**) sera redirigée vers un back-end, de manière round robin (voir tests plus loin), vers l’un des 3 backend ayant les adresse IP : 172.17.0.3 … jusqu’à 172.17.0.1, sur le port 80 en modifiant l’URI de la requête en mettant un / (c’est pour cette raison que c’est marqué ProxyPass /api balancer://backend **/**

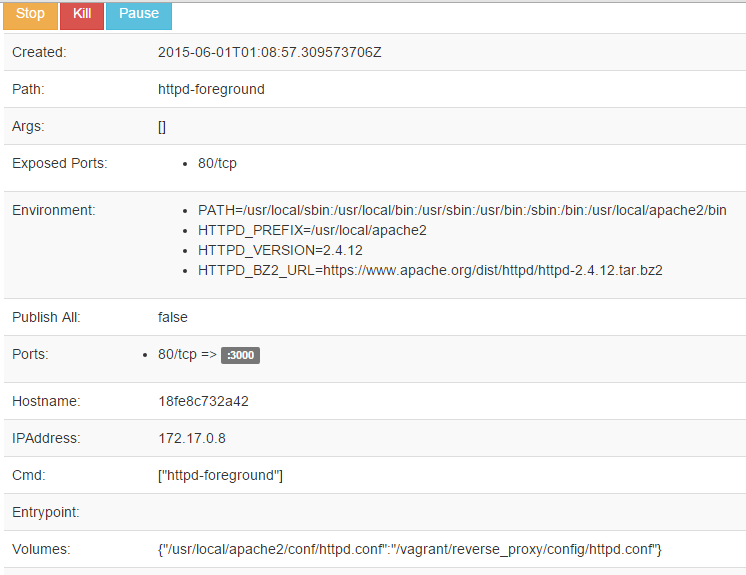
C’est pareil pour le front-end, sauf que la méthode de répartiteur de charge dépend du traffic. Il y a en plus de ça une route associé à chaque container front-end. Par exemple le front end qui a l’adresse IP 172.17.0.5 est sur la route numéro 2. Ce numéro de route est envoyée au client dans un cookie grâce à une entête Set-Cookie « Route ID = 2 » par exemple. De cette manière, lorsque le client enverra une requête http avec une URI **/**, il accèdera à un front-end aléatoire s’il n’a pas de cookie, et ensuite la prochaine fois qu’il se connecte il y aura le numéro 2 comme route dans son cookie, et donc c’est le même front-end qui répondra tout le temps (celui qui est sur la route 2 donc 172.17.0.5.

Voici une démonstration :

docker run **-v "/vagrant/reverse\_proxy/config/httpd.conf:/usr/local/apache2/conf/httpd.conf"** -d -p 3000:80 reverseproxy

*La partie en gras permet de monter un volume vers le fichier httpd.conf que l’on a modifié.*

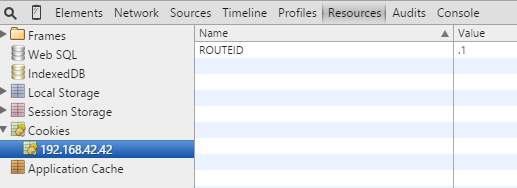
On lance donc un container reverseproxy :



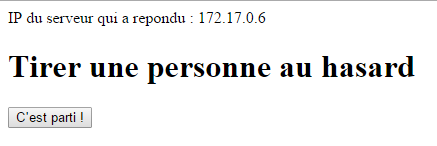
Ensuite on se connecte à l’URL 192.168.42.42:3000 et on doit voir apparaître une page html provenant du front end :



On voit que le front-end qui a répondu est le 172.17.0.6 et on voit dans la réponse http le cookie avec le numéro de la route :

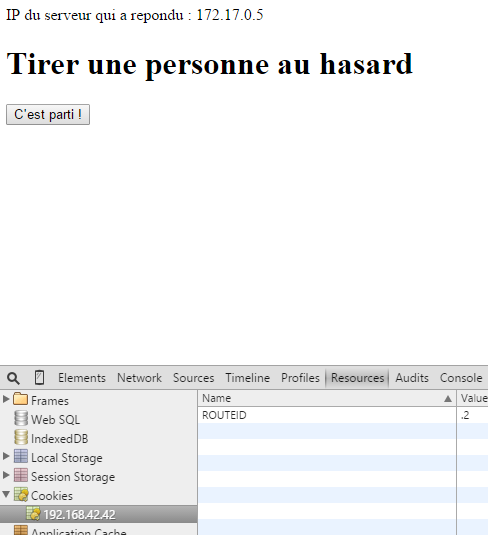


On voit que le client a reçu un cookie avec le numéro de route 1, donc si on rafraichit la page c’est le même frontend qui doit répondre.



C’est le cas car on voit que c’est le même qui a répondu.

Si on efface les cookies, c’est normalement un autre frontend qui doit répondre :



C’est donc celui qui a la route 2 dans le httpd.conf qui a répondu soit 172.17.0.5

Si on clique sur le bouton, on doit recevoir un étudiant au hasard :



On voit que les backend on fait leur travail car il ont envoyé des étudiants (à noter que nous avons cliqué plusieurs fois sur le bouton).

Après plusieurs tentatives Wireshark ou tcpdump, nous ne sommes pas arrivés à montrer que c’est à chaque fois un backend différent qui répond, car les IP affichées sont seulement celle de vagrant ou de la machine physique.

## Hearbeat front-end et back-end

### Heartbeats de NodeJS

Durant le cours nous avons beaucoup parlé de NodeJS ansi que du terme HearthBeat ce qui nous a tournés vers le module NodeJS nommée Heartbeat disponible ici :

<https://www.npmjs.com/package/heartbeats>

Le module semblait faire ce que nous voulions, malheureusement il n’est pas bloquant est notre scripte JS s’arrêtait. Une solution plus standard a donc été envisagée.

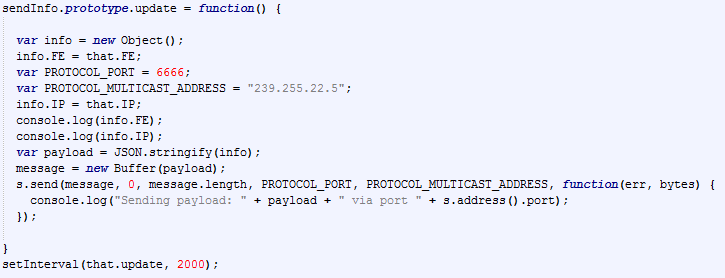
### Heartbeat

Petit définition pour commencer :

*« In computer science, a heartbeat is a periodic signal generated by hardware or software to indicate normal operation or to synchronize other parts of a system. Usually a heartbeat is sent between machines at a regular interval on the order of seconds. If a heartbeat isn't received for a time—usually a few heartbeat intervals—the machine that should have sent the heartbeat is assumed to have failed. »*

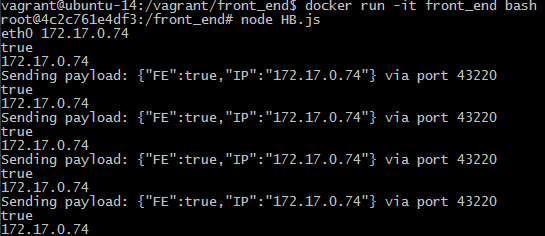
L’intérêt du heartbeat dans notre cas est d’informer le revers proxy de notre existence en lançant un broatcast toute les X seconde et en disant qui nous sommes et ce que nous savons faire. Dans notre cas le front\_end enverra son adresse id ainsi que un boolean valant « true » indiquant qu’il s’agit d’un front\_end, il en va de même dans le cas du back\_end mais celui-ci envera « false » indiquant qu’il ne s’agit pas du front\_end, par déduction nous avons qu’il s’agit du back\_end.

Dans notre cas nous envoyons un objet sérialisé en json dans un datagram udp, cette objet serialisé contient l’ip de notre machine lançant le scripte JS ainsi que un « true » dans le cas du front\_end et un « false » dans le cas du back\_end, et cela toutes les deux seconde en broatcast.

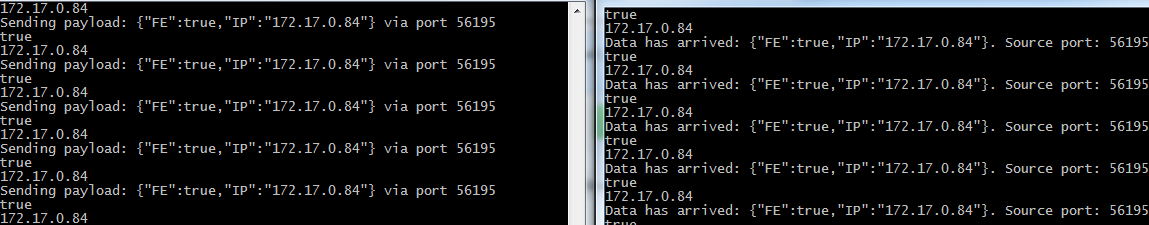


### Test

Afin de démonter et de tester le bon fonctionnement de notre heartbeat nous lançons notre front end en interactif suivit de « bash » pour ouvrir un bash. Nous pouvons observer que un message avec l’ip de notre machin et un « true » sont envoyé sous la forme d’un objet serialisable.



Ici nous observons qu’il y a bien une communication qui se crée entre le front\_end et le contrôleur du revers proxy. L’objet sériablisable est bien récupéré par le contrôleur. Malheureusement cela ne fonctionne que si on lance nos conteneurs en interactif, pour des raisons inconnues l’ip n’est pas reçu par le contrôleur si nous lançons nos conteneur en daemon.



# Conclusion

## Simon Baehler

Ce laboratoire fût un peu comme naviguer en plein brouillard, on ne sait pas où l’on va, alors on va à l’instinct. Ce laboratoire a été sans doute le plus intéressant des laboratoires que nous avions à faire car il y avait vraiment quelque chose de concret.

## Raphaël Racine

Ce laboratoire a été pour moi très intéressant, notamment la partie reverse proxy, que je ne connaissais pas avant.

Je regrette qu’on ait pas eu le temps de nous occuper de la modification du fichier httpd.conf du reverse proxy, ceci étant dû à des problèmes de communication dans le groupe, mais au moins j’aurais à peu près su comment le faire, mais par manque de temps nous n’avons pas pu réaliser entièrement la partie UDP, mais nous avons au moins écrit un heartbeat du côté back-end aussi bien que dans le côté front-end.

Si on avait eu un peu plus de communication dans le groupe, on aurait pu finir.

## Armand Delessert

## Olivier Djeulezeck