

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 | lundi 1er juin 2015 |

# Table des matières

[Table des matières 1](#_Toc420924133)

[1. Objectifs 2](#_Toc420924134)

[2. Architecture 3](#_Toc420924135)

[3. Développement 4](#_Toc420924136)

[3.1. Back-end 4](#_Toc420924137)

[3.1.1. Structure des fichiers 4](#_Toc420924138)

[3.1.2. Dockerfile 5](#_Toc420924139)

[3.1.3. Start.sh 5](#_Toc420924140)

[3.1.4. Test 6](#_Toc420924141)

[3.1. Front-end 7](#_Toc420924142)

[3.1.1. Structure des fichiers 7](#_Toc420924143)

[3.1.2. Dockerfile 8](#_Toc420924144)

[3.1.3. Start.sh 9](#_Toc420924145)

[3.1.4. Test 9](#_Toc420924146)

[3.2. Reverse proxy 11](#_Toc420924147)

[3.2.1. Structure des fichiers 11](#_Toc420924148)

[3.2.2. Load Balancer 11](#_Toc420924149)

[3.2.3. Dockerfile 12](#_Toc420924150)

[3.2.4. Test 12](#_Toc420924151)

[3.3. Hearbeat du front-end et du back-end 20](#_Toc420924152)

[3.3.1. Heartbeats de NodeJS 20](#_Toc420924153)

[3.3.2. Heartbeat 20](#_Toc420924154)

[3.3.3. Test 21](#_Toc420924155)

[4. Conclusion 22](#_Toc420924156)

[4.1. Simon Baehler 22](#_Toc420924157)

[4.2. Raphaël Racine 22](#_Toc420924158)

[4.3. Armand Delessert 22](#_Toc420924159)

[4.4. Olivier Djeulezeck 22](#_Toc420924160)

# Objectifs

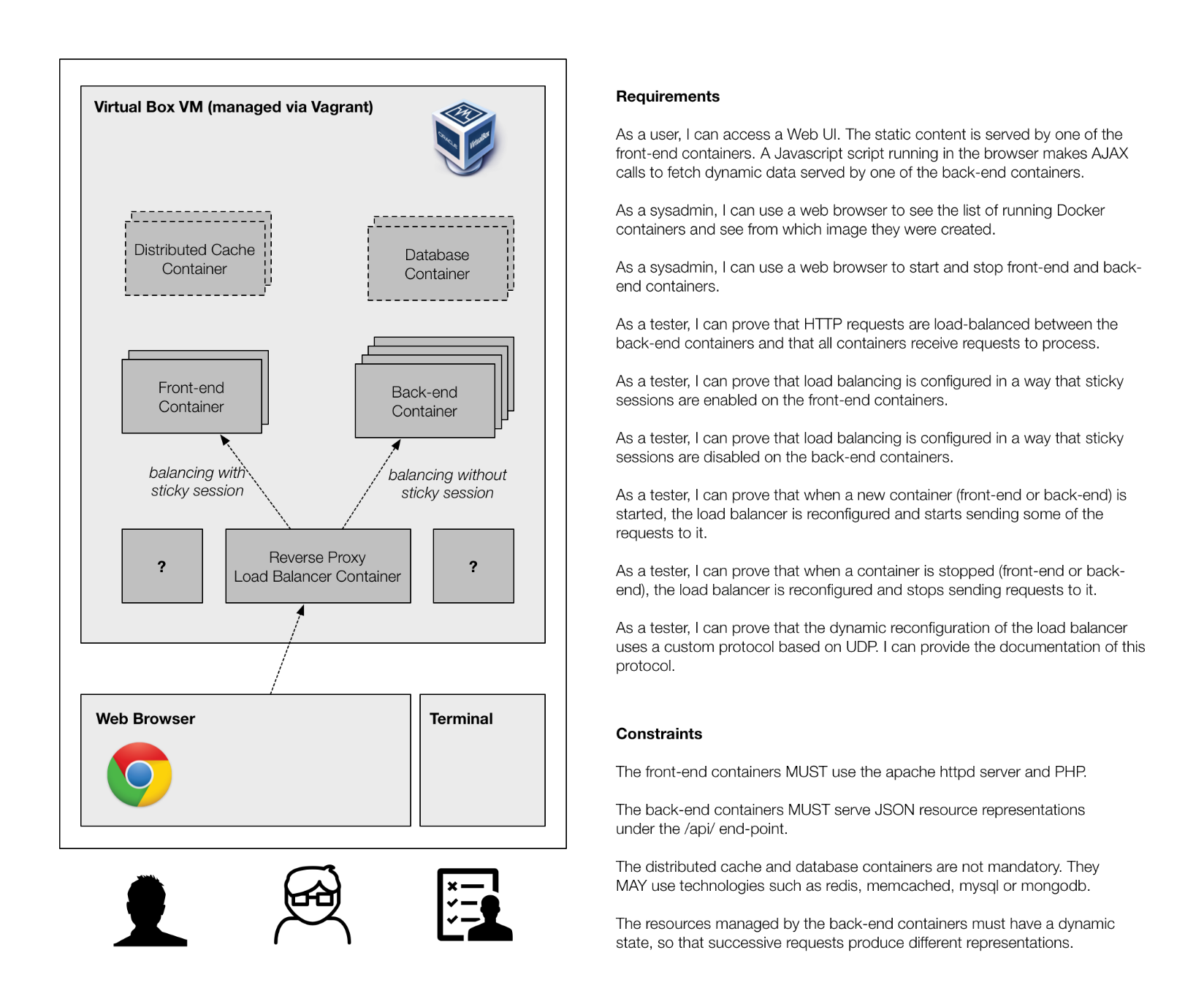
L’objectif principal de ce labo est d’expérimenter la mise en place d’une infrastructure web et de se familiariser avec les différents composants de celle-ci (serveurs http, proxys inverses, équilibreurs de charge).

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://192.168.42.42:3000/>) 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://192.168.42.42:3000/](http://exemple.com/)api](http://192.168.42.42:3000/api).

# Développement

## 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 « / », 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 » :

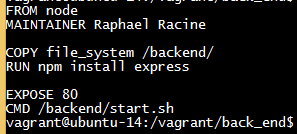


Contenu du répertoire « file\_system » :



### 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 Docker.

Ensuite, on fait une installation de « express » 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. « start.sh » 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 est démarré. Voici le contenu de ce fichier :

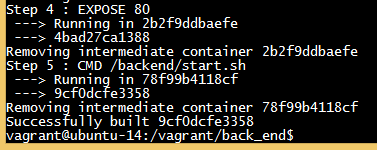


On démarre l’outil « node » en lançant d’abord le script « back\_end.js » (qui se trouve dans le répertoire « backend » qui avait été copié dans l’image), et on fait la même chose pour le « HB.js ».

### Test

Tout d’abord, on construit une image nommée « backend » en tapant en ligne de commande :

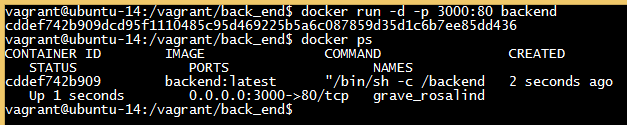
docker build –t backend .



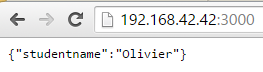
L’image a bien été créée.

Ensuite, on lance notre « back-end » grâce à la commande :

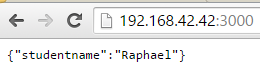
docker run –d –p 3000:80 backend



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 entrant simplement l’IP de la machine Vagrant dans le navigateur web (<192.168.42.42:3000>).



On voit que le « back-end » 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.

La partie « heartbeat » sera détaillée dans une section dédiée plus loin.

## Front-end

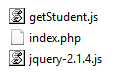
Nous avons décidé d’écrire une page PHP (« index.php ») qui possède un bouton sur lequel on cliquera dessus pour envoyer une requête AJAX à un serveur « back-end » via le « reverse proxy » (détaillé plus loin). Le « reverse proxy » s’occupera de rediriger la requête HTTP qui contient l’URI « /api » vers un des « back-end » en modifiant l’URI en « / ». La modification de l’URI est nécessaire car le « back-end » traite les requêtes HTTP qu’il reçoit sur l’URI « / ».

### Structure des fichiers

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

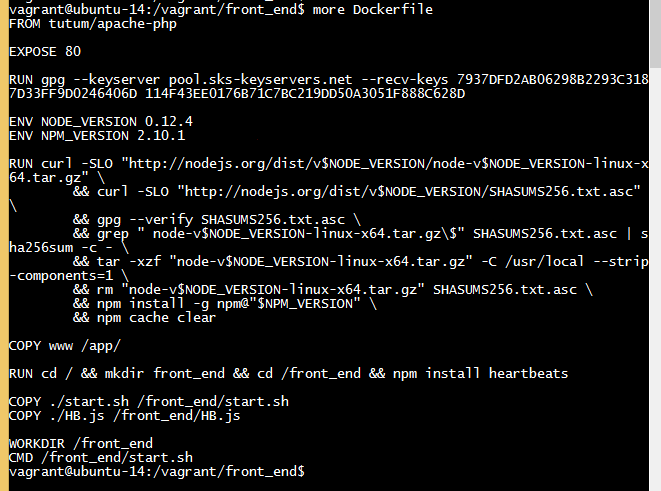


Contenu du répertoire « www » :



### 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 front-end qui répond.

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

Toutes les instructions d’après (jusqu’au COPY) permettent d’installer Node.js pour lancer le script « HB.js ».

Ensuite, on copie le répertoire « www » dans « /app/ » dans le container. C’est à partir de ce répertoire que 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. Voici le contenu de ce fichier :

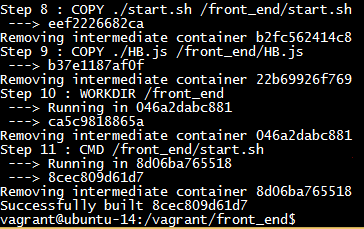


On démarre d’abord le serveur apache et ensuite le script « HB.js » avec l’utilitaire « node ».

### Test

Tout d’abord, on construit une image nommé front-end en tapant en ligne de commande :

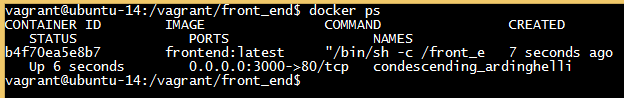
docker build –t frontend .



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 entrant simplement l’IP de la machine Vagrant dans le navigateur web (<192.168.42.42:3000>).



On voit que le front-end a bien répondu en envoyant du contenu HTML.

## Reverse proxy

Pour cette partie, nous avons décidé d’installer 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 Docker, ceci afin de pouvoir modifier ce fichier depuis le controlleur du heartbeat qui se trouverait dans un autre container Docker.

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

### 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  (seule la partie concernant le reverse-proxy est montrée ici) :



Voir les explications dans la section Test.

### Dockerfile

Voici le contenu du Dockerfile concernant la partie reverse proxy :

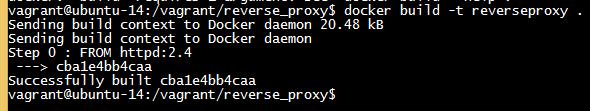


On se base simplement sur l’image Docker nommée « httpd:2.4 », pour avoir un serveur Apache fonctionnel.

### 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 créée.

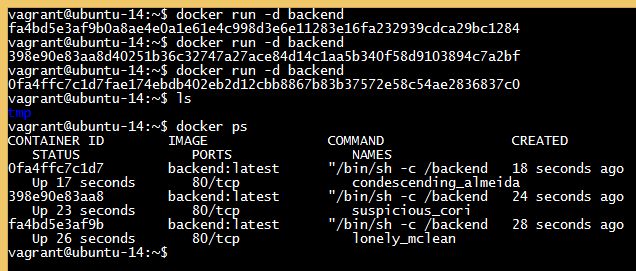
Nous n’avons malheureusement pas eu le temps de faire la partie contrôleur pour éditer dynamiquement le fichier « httpd.conf » du reverse proxy lorsque des containers démarrent ou s’arrêtent. Nous devons donc lancer d’abord les back-end, suivi des front-end, et enfin le reverse proxy. Nous devons lancer le reverse proxy à la fin en montant un volume vers le fichier « httpd.conf » après avoir introduit les adresses IP des containers front-end et back-end. En effet, Docker attribuant à chaque fois une adresse IP différente aux nouveaux containers Docker, nous sommes obligés de commencer par démarrer les containers front-end et back-end pour connaître leur adresses IP et les entrer dans le fichier de config du reverse proxy.

**En pratique, nous devons suivre les points suivants pour pouvoir démarrer le système :**

Démarrage des back-end

On commence par créer par exemple 3 serveurs back-end en lançant 3 fois la commande :

docker run –d backend

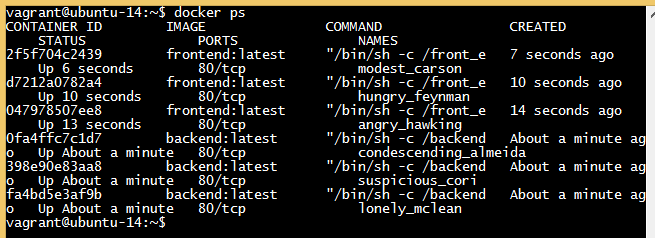


On a donc 3 serveurs back-end qui écoutent sur le port 80.

Démarrage des front-end

On créer également par exemple 3 serveurs front-end en lançant 3 fois la commande :

docker run –d frontend



On a donc 3 serveurs front-end qui écoutent sur le port 80.

Démarrage du reverse proxy

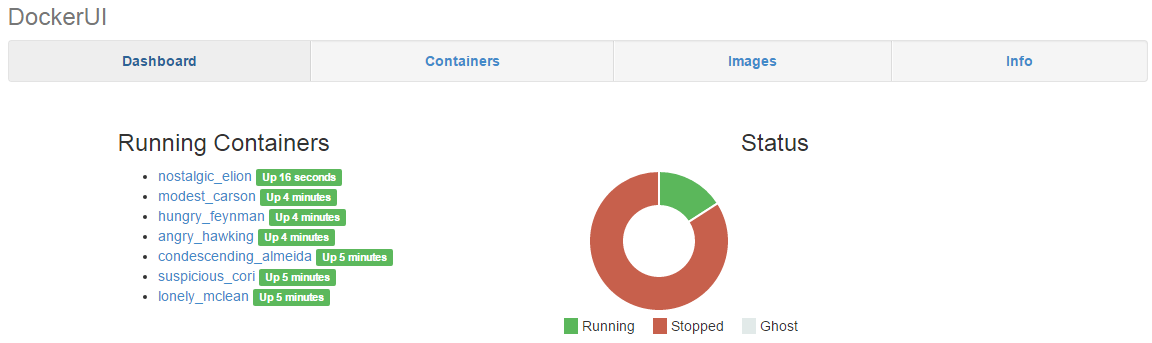
Avant de démarrer le reverse proxy, nous devons compléter le fichier « httpd.conf » avec les adresses IP des containers front-end et back-end lancés plus tôt. Pour cela, nous allons nous rendre sur l’interface Docker UI pour récupérer les adresses IP des containers Docker en cours d’exécution.

Entrer la commande suivante pour lancer Docker UI si celui-ci ne s’exécute pas déjà :

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

Docker UI est une interface web pour Docker. Elle va nous permettre de visualiser facilement les containers Docker en cours d’exécution et ainsi récupérer leurs adresses IP.

On se connecte sur Docker UI dans un navigateur à l’aide de l’URL <192.168.42.42:9000>.



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

Par exemple, pour front-end nous voyons la page suivante :



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

Il nous faut donc modifier notre fichier « httpd.conf » qui se trouve dans le reverse proxy dans le dossier « reverse\_proxy/config » en configurant la partie Load Balancer de la manière suivante :



Dans l’encadré **orange** on a mis une règle concernant le back-end.

Cela signifie que chaque requête qui arrive sur l’URI « /api » (selon le contenu de l’encadré **vert**) sera redirigée vers un des serveurs back-end avec comme URI : « / » (voir tests plus loin). C’est pour cette raison que c’est marqué ProxyPass /api balancer://backend/.

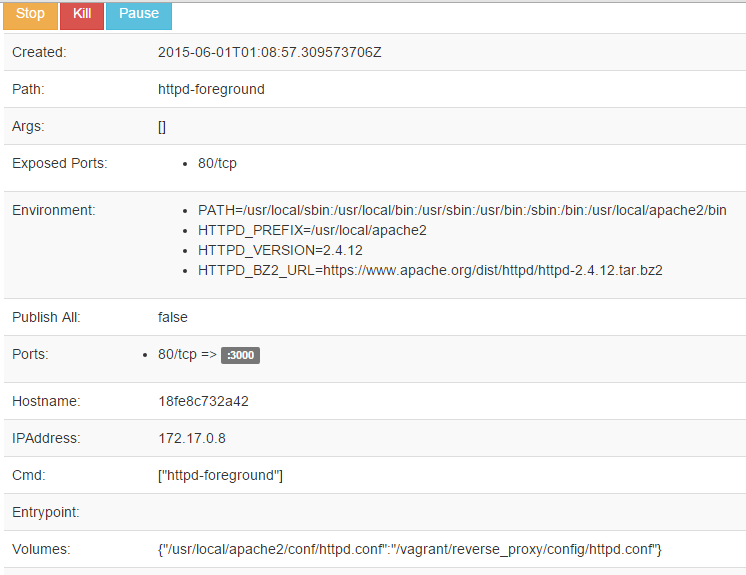
Pour la partie front-end, nous procédons de manière identique, sauf que la répartition de la charge dépend du trafic. Il y a en plus de ça une route associé à chaque container front-end. Par exemple le front-end qui a l’adresse IP 172.17.0.5 est sur la route numéro 2. Ce numéro de route est envoyée au 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 l’URI « / », il accèdera à un front-end aléatoire s’il n’a pas de cookie et la prochaine fois qu’il se connectera, il y aura le numéro 2 comme route dans son cookie. C’est donc le même front-end qui répondra à chacune de ses requêtes (celui qui est sur la route 2 donc à l’adresse IP 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 (-v "…") permet de monter un volume vers le fichier « httpd.conf » que l’on a modifié avant le lancement du reverse proxy.*

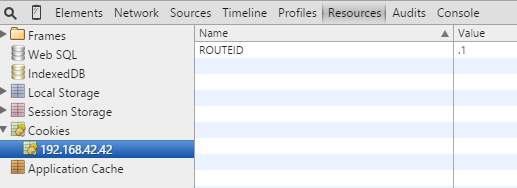
On lance donc un container « reverseproxy » :



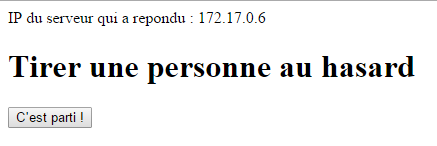
Ensuite, en se connectant avec un navigateur web à l’URL 192.168.42.42:3000, une page HTML provenant du front-end doit apparaître :



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 rafraîchit la page c’est le même front-end qui doit répondre.



Et c’est bien le même qui a répondu.

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



C’est donc celui qui a la route 2 dans le « httpd.conf » et qui possède l’adresse IP 172.17.0.5 qui a répondu.

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



On voit que les back-end on fait leur travail car ils ont envoyé des noms d’étudiants. À noter que nous avons cliqué plusieurs fois sur le bouton pour obtenir une liste de noms.

Après plusieurs tentatives Wireshark ou tcpdump, nous ne sommes pas arrivés à montrer que c’est à chaque fois un back-end différent qui répond, car les IP affichées sont seulement celle de Vagrant ou de la machine physique et non celles des containers Docker.

## Hearbeat du front-end et du back-end

### Heartbeats de NodeJS

Durant le cours nous avons beaucoup parlé de NodeJS ainsi que du terme hearthbeat ce qui nous a tournés vers le module NodeJS nommé « Heartbeat » disponible à l’adresse suivante :

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

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

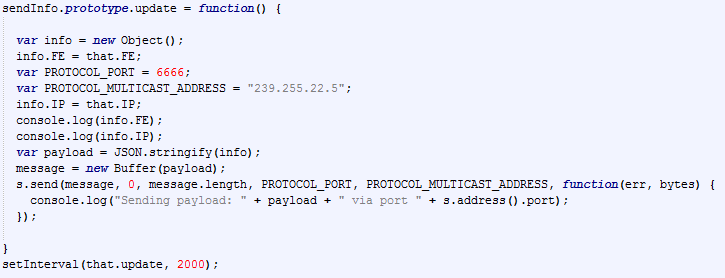
### Heartbeat

Voici une petite 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. »*

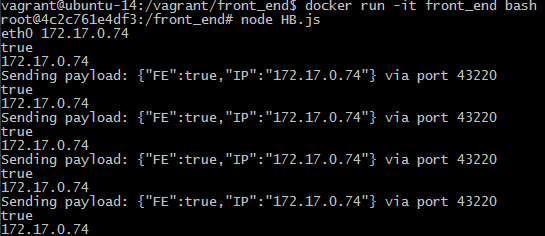
L’intérêt du heartbeat dans notre cas est d’informer le revers proxy de notre existence en lançant un broatcast toute les X secondes et en disant qui nous sommes et ce que nous savons faire. Dans notre cas le front-end enverra son adresse IP ainsi qu’un booléen valant « vrai » pour indiquer qu’il s’agit d’un front-end. Il en va de même dans le cas du back-end mais celui-ci enverra un booléen « faux » indiquant qu’il ne s’agit pas d’un front-end mais d’un back-end.

Dans notre cas nous envoyons un objet sérialisé en JSON dans un datagramme UDP. Cet objet sérialisé contient l’IP de notre machine exécutant le scripte JavaScript ainsi qu’un booléen « vrai » dans le cas du front-end et un booléen « faux » dans le cas du back-end. Cet objet est envoyé toutes les deux seconde en broatcast.

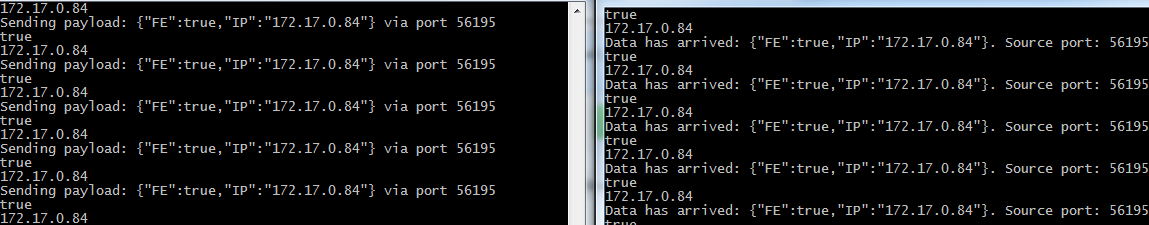


### Test

Afin de tester et de démontrer le bon fonctionnement de notre heartbeat nous lançons notre front-end en interactif suivit de « bash » pour ouvrir un Bash. Nous pouvons observer qu’un message avec l’IP de notre machin et un booléen « vrai » sont envoyé sous la forme d’un objet sérialisable.



Ici nous observons qu’il y a bien une communication qui se passe entre le front-end et le contrôleur du revers proxy. L’objet sérialisable est bien récupéré par le contrôleur. Malheureusement cela ne fonctionne que si on lance nos containers en interactif. Pour des raisons inconnues l’IP n’est pas reçue par le contrôleur si nous lançons nos containers en tant que 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 n’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

Sincèrement, je suis étonné du résultat obtenu. Au début, je ne pensais pas qu’on arriverait à finir ce laboratoire. La donnée ne fournissait aucune indication de la méthode à suivre et aucun code de base n’était fourni. Nous avions juste un script JavaScript d’exemple vu brièvement quelques leçons plus tôt. Heureusement que beaucoup de périodes de laboratoire et de cours étaient réservées pour ce labo sans lesquelles nous n’y serions jamais arrivés. Finalement le résultat obtenu fait plaisir à voir et ce laboratoire permet de bien comprendre les contraintes de charge des architectures des services web.

## Olivier Djeulezeck