

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

[1. Objectifs 2](#_Toc420485598)

[2. Architecture 3](#_Toc420485599)

[3. Développement 4](#_Toc420485600)

[3.1. Reverse proxy 4](#_Toc420485601)

[3.2. Front end 4](#_Toc420485602)

[3.3. Back end 4](#_Toc420485603)

[3.4. Heartbeats du front end et du back end 4](#_Toc420485604)

[3.5. Load balancer 4](#_Toc420485605)

[4. Procédure de validation 4](#_Toc420485606)

# Objectifs

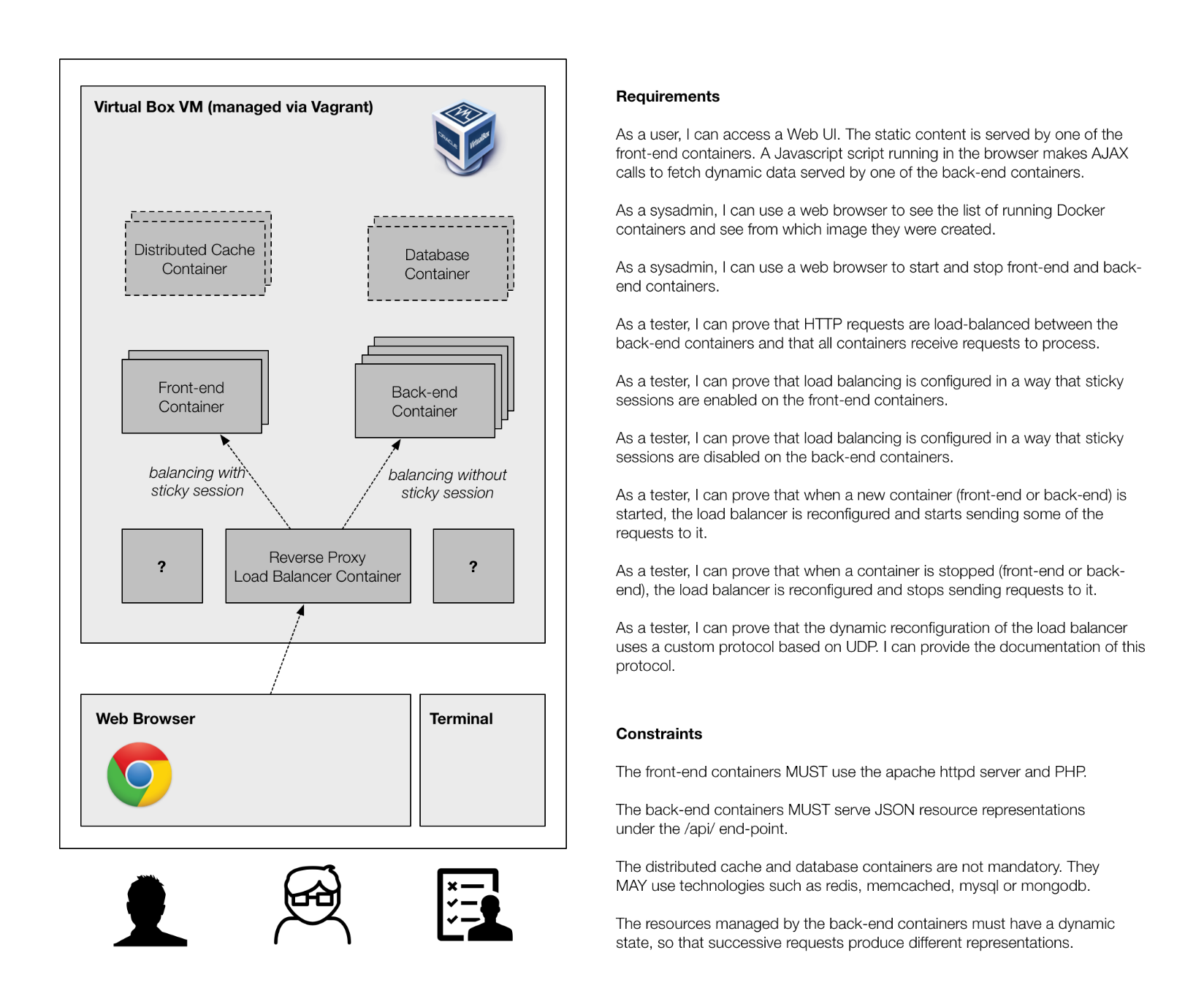
L’objectif principal de ce labo est d’expérimenter la mise en place d’une infrastructure web et de se familiariser avec les différents composants de celle-ci (serveurs http, proxys inverses, équilibreurs de charge).

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 »

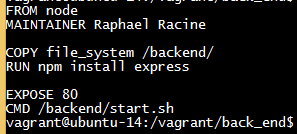


Contenu du répertoire file\_system



### 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

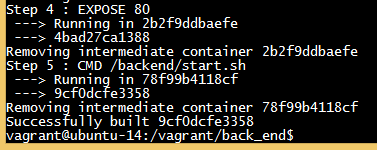
Les commandes qui se trouvent dans ce fichier shell sont lancées lorsque le container et démarré. Le contenu de ce fichier est :



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.

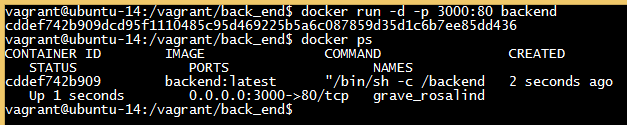
### Test

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

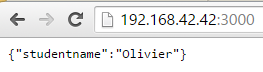


L’image a bien été créé.

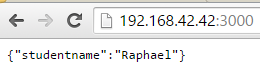
Ensuite, on lance notre back-end grâce à la commande   
docker run –d –p 3000:80 backend



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.

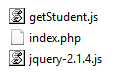
Nous avons décidé d’écrire une page php (index.php) qui possède un bouton sur lequel on cliquera dessus pour envoyer une requête AJAX à un backend (via le reverse proxy, voir plus loin) http avec l’URI ‘/api’ que le reverse proxy redirigera vers un des backend avec l’URI ‘/’, comme le backend traite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

### Structure des fichiers

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

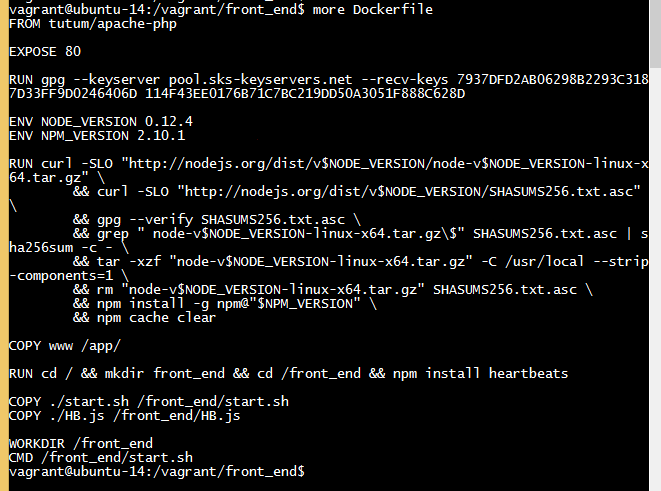


Contenu du répertoire www



### 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.

**SIMON 🡪 EXPLIQUER LA LIGNE OU CEST MARQUER RUN…**

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

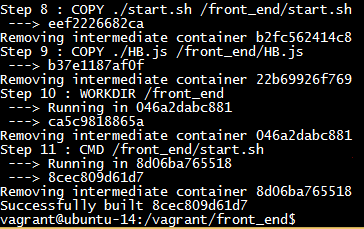
Les commandes qui se trouvent dans ce fichier shell sont lancées lorsque le container et démarré. Le contenu de ce fichier est :



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

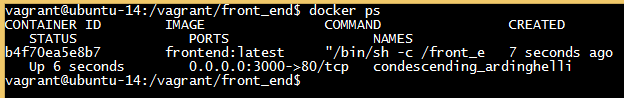
### Test

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



L’image a bien été créé.

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

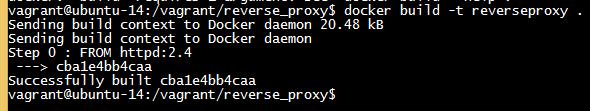
Voici le contenu du Dockerfile concernant la partie reverse proxy :



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, nous allons lancer 3 container…**

## Hearbeat front-end et back-end

# Conclusion