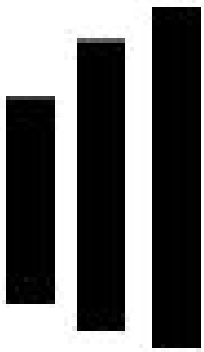
Transcode

Docker Based Transcoding infrastructure

4PJT - Transcode



159979 Wassim DHIF - 159024 Samy Laumonier - 158390 Nicolas Rouvière - 159754 Jordan Cousserans - 159172 Lilian Guillot

Sommaire

Software Development

Web Client

Concernant notre Web Client, nous utilisons un environnement Full Stack JavaScript. Nous combinons Meteor qui est un framework basé sur NodeJS et Blaze, le moteur de template officiel de Meteor.

Ce choix s'explique par les performances de base de NodeJS et la facilité de développer en Meteor, une technologie très adaptée pour les web app.

Au niveau base de données, nous utilisons MongoDB ainsi que MiniMongo nous permettant de communiquer entre base côté serveur et base côté client de façon très simple.

Grâce aux websockets parfaitement intégrés avec NodeJS et Meteor, les données sont actualisées en temps réel. Ce qui est très pratique pour une web app comme Transcode nécessitant une réactivité très importante.

Utilisateurs

Meteor intègre des packages officiels comme les gestion des utilisateurs (connexion, inscription...).
Ce package s'appelant "accounts" intègre des adons permettant la connexion OAUTH à Google, Facebook et bien d'autres.

Une fois installé, il nous a suffit de "render" un composant du package dans notre navbar.

Grâce une interface très détaillée, le package nous indique toutes les étapes pour configurer nos applications/services chez Google et Facebook.

Il nous a simplement fallu créer quelques helpers pour gérer les différences entre les services (password, google ou facebook), notamment pour les emails.

Pour finir, nous avons créer une route pour le profil utilisateur grâce au package Iron.Router, où nous nous abonnons à la publication qui récupère les données de l'utilisateur, ainsi que ses fichiers et ses tâches lancées.

Upload de fichiers

Sur son profil, l'utilisateur peut upload des fichiers depuis son ordinateur ou depuis un lien externe.

Nous utilisons un package qui gère le formulaire d'upload et les événements le concernant.

De manière interne, le package vérifie grâce à une regex le type du fichier, puis il le charge dans un dossier que nous avons spécifié.

Nous appelons donc un callback lorsque le fichier a terminé d'être téléchargé.

Ce callback se charge de créer un nouveau document dans la collection Files grâce aux données extraites du fichier par FileSystem (FS), composant interne à NodeJS.
Avec FS nous récupérons la taille du fichier, son nom, son type..., puis nous renommons le fichier original pour lui donner un identifiant unique.
Très important, nous stockons le chemin du fichier dans le document.

Une fois le document créé, nous l'affichons dans un tableau détaillé.
Chaque ligne du tableau correspond à un fichier que l'on peut convertir ou supprimer.

Mais encore, nous utilisons des hooks qui, à chaque upload ou suppression de fichier, incrémentent ou décrémentent l'usage de mémoire de l'utilisateur (limité à 10GB).

Pour finir, nous avons géré la sécurité de ces fichier grâce au package security qui refuse toutes manipulations (créations ou suppression) de fichiers ne nous concernant pas.

Paiements

Lorsqu'un utilisateur clique sur “convertir” un fichier, il peut choisir dans un champ de type select le format vers lequel il veut convertir son fichier.

Une fois le format choisi, une fenêtre de paiement apparaît lui indiquant le prix à payer pour convertir son fichier.

Nous calculons le prix en fonction de la taille du fichier.

Nous transmettons ce prix à l’API Stripe qui s’occupe d'effectuer les transactions.

Nous avons préféré Stripe car leurs commissions sont inférieures à celles de Paypal. De plus, ils gèrent les paiements directs (par carte de crédit) sans sortir du domaine de Transcode, ce qui manque à PayPal, notamment à cause de leur politique internationale.

Une fois le paiement effectué, nous modifions le document Files pour lui mettre le statut “processing” et nous créons un nouveau document Tasks contenant l’id du fichier à convertir et le format vers lequel convertir.

Communiquer avec le core

À ce point, il faut pouvoir communiquer entre le web client et le core (la partie qui s'occupe de convertir les fichiers.

Pour cela, nous utilisons la technologie Celery qui s'occupe d'ordonnancer les tâches et de répartir la charge sur les workers.

Celery est donc notre intermédiaire, il faut pouvoir lui parler. Nous n'interagissons jamais directement avec FFMpeg dans le core.

De ce fait, nous utilisons un client Celery pour Node.js :
https://github.com/mher/node-celery

Celui-ci se connecte à Celery et lui transmet les tâches à ordonnancer.
Une fois la tâche complètée, il exécute un callback qui prend en paramètre le résultat du core, c'est à dire le chemin du fichier convertit.

Ce callback s'occupe de changer le statut du document File lié à la tâche traitée (de "processing" à "converted"), puis il remplace le fichier original par le fichier converti.

Au niveau du dashboard, l'utilisateur peut dorénavant télécharger le fichier qui vient d'être converti.

Le core

Le core, réalisé en python, est la partie qui s'occupe de convertir les fichiers de manière optimale.

Il travaille en collaboration avec Celery qui ordonnance le travail (les workers) et FFmpeg qui permet d'extraire des sous-parties du fichier pour les traitées sur les différents workers (les convertir) et les concaténer par la suite.

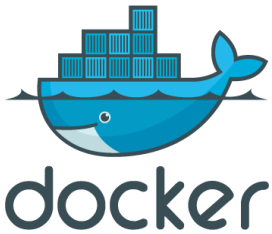
Celery est un ordonnanceur de tâche asynchrone open source basés sur la distribution de message.
Les conteneurs celery contiennent FFMPEG qui est en charge du traitement du flux audio et video. FFMPEG est au coeur du processus de transcoding.

1. FFmpeg définit les intervales à traiter séparemment (il ne découpe pas le fichier)
2. Celery s'occupe de lancer la conversion des parties
    a. Il inspecte quels workers sont libres
    b. Il leur dit à partir quand il faut traiter et à partir de quand il faut s'arrêter
    c. Une fois toutes les parties traitées, il ordonne à FFmepg de les concaténer
3. Une fois le fichier concaténé, il est totalement converti, la tâche Celery prend donc fin et renvois le chemin du nouveau fichier

Au moment où la tâche Celery terminée, notre client pour Node.JS appelle notre callback comme expliqué plus haut.

Supporting Architecture

Pour mener à bien la partie infrastructure du projet transcode nous avons opté pour une approche DevOps : le développement d'infrastructure ou Infrastructure as Code ou IaC. L'IaC permet une grande adaptabilité de notre infrastructure en automatisant la mise en place et le déploiement d'infrastructure. De plus l'adressage des IPs est automatique ce qui facilite le networking.



Docker est un logiciel libre qui automatise le déploiement d'applications dans des conteneurs logiciels. Son fonctionnement est simple. Il empaquette une application et ses dépendances dans un conteneur virtuel. Ce principe nous permet d'exécuter des applications sur n'importe quel os linux.

Dans le cadre du projet transcode l'utilisation de docker nous a permis de déployer facilement d'autre conteneur avec différentes fonctions : WebServer Meteor, Core, Worker

De plus grâce à Swarm l'infrastructure peut facilement être déployée sur plusieurs noeuds.

Nous divisons l'architecture en deux parties : d'un côté meteor, balancers et mongoDB, de l'autre le core et les workers.

Balancers

Les balancers permettent à la fois de répondre à une charge trop importante du service en la répartissant sur plusieurs serveurs, et de réduire l'indisponibilité potentielle de ce service que pourrait provoquer une erreur.

Le HAProxy se connecte à docker avec les sockets de docker pour voir les containers Meteor qui sont UPs et les rajouter.

Containers Meteor

Les containers Meteor sont des containers Docker qui utilisent l'image MeteorD.
Ils permettent de faire tourner notre web client de manière optimale.

De plus, nous utilisons **Meteor Cluster** qui s'occupe de repartir les charges sur les différents web servers.

MeteorD

Comme vu plus haut nous avons utilisé le framework Meteor pour la partie web client.
Pour cela nous utilisons l'image MeteorD par MeteorHacks.

MeteorD est une image Docker permettant de build à la volé son application sous la forme d'une image Docker qui sera compatible avec Compose.
Ce choix s'avère donc totalement en accord avec notre approche IaC.
Nous modifions également le Dockerfile pour l'adapter a nos besoins.

```
FROM meteorhacks/meteord:onbuild
MAINTAINER Wassim DHIF <wassimdhif@gmail.com>

ONBUILD COPY ./ /app
ONBUILD RUN bash $METEORD_DIR/on_build.sh

RUN \
        echo "deb http://repo.mongodb.org/apt/debian wheezy/mongodb-org/3.0
main" | tee /etc/apt/sources.list.d/mongodb-org-3.0.list && \
        apt-get update && apt-get upgrade -y

RUN apt-get install -y --force-yes mongodb-org-shell
RUN npm install -g download-file fibers hiredis mime node-celery

COPY ./wait-mongo.sh /wait-mongo.sh

EXPOSE 80

ENTRYPOINT bash /wait-mongo.sh
```

Nous commençons par build notre application avec les instructions ONBUILD.
Nous ajoutons ensuite le shell mongo qui va nous permettre de vérifier si nous container mongo sont prêt. Nous installons également les modules NPM nécessaire à l'application.
On ajoute notre script wait-mongo.sh, on expose le port 80 et on démarre l'application.

MongoDB

wait-mongo.sh

```
#!/bin/bash

set -e

id=1
members=()
IFS=',' read -ra hosts <<< "${PRIMARY_MEMBER},${SECONDARY_MEMBERS}"
for host in "${hosts[@]}"; do
   members+=("{_id:${id},host:'${host}'}")
   ((id++))
done
members_js=`echo $(IFS=,; echo "${members[*]}")`
js="rs.initiate({_id:'${REPLICA_SET_ID}',members:[${members_js}]});"

until mongo --host
"${REPLICA_SET_ID}/${PRIMARY_MEMBER},${SECONDARY_MEMBERS}"
"${DATABASE}"; do
  >&2 echo -e "\033[1m mongo --host
${REPLICA_SET_ID}/${PRIMARY_MEMBER},${SECONDARY_MEMBERS}
${DATABASE}"
  >&2 echo -e "\033[1m Mongo is unavailable - sleeping"
  sleep 1
done

>&2 echo -e "\033[1m Mongo is up - Starting meteor"
chmod +x $METEORD_DIR/run_app.sh
exec $METEORD_DIR/run_app.sh
```

Nous récupérons la liste des membres MongoDB en variable d'environnement, nous l'utilisons alors pour construire la MONGO_URL et enfin nous testons avec le shell Mongo pour quand le ReplicaSet MongoDB est prêt, une fois que la base de données est prête, nous lançons l'application.

Pour faire de la réplication avec mongoDB nous avons utilisé la technologie de ReplicaSet. Elle permet d'effectuer une réplication entre au moins 3 instances MongoDB. Notre infrastructure contient donc 3 conteneurs MongoDB, qui vont être configurer par un quatrième conteneur qui va se charger de configurer les 3 autres conteneurs.

RabbitMQ

RabbitMQ est le broker qui permet de faire la liaison entre tous les containers de Celery (cores et workers).

Les Celery communiquent impérativement par RabbitMQ, c'est la colonne vertébrale de la communication entre les containers.

Les cores

Les cores sont des containers pour Celery et FFMpeg.
Comme vu plus haut, ils s'occupent de convertir de manière ordonnées les fichiers

Celery est l'ordonanceur, c'est celui qui gère les tâches.
FFMpeg est le convertisseur, c'est celui qui analyse puis qui traite les fichiers.

Le core communique avec le web client grâce au client Celery pour Node.js

Les workers

Les workers sont exactement les mêmes containers que les cores.
Ils contiennent aussi Celery et FFMpeg, mais n'ont pas le même point d'entrée.

En effet, sur le core, FFMPeg s'occupe seulement de concaténer les parties alors que les fichiers sont traités seulement sur les workers.

Résumé de l’infrastructure