Université de Caen Basse Normandie
Système - François Rioult

Rapport de projet MusicBrainz

Suyin LEFEBVRE et Anthony ROBIN

15 janvier 2014

Sommaire

Introduction

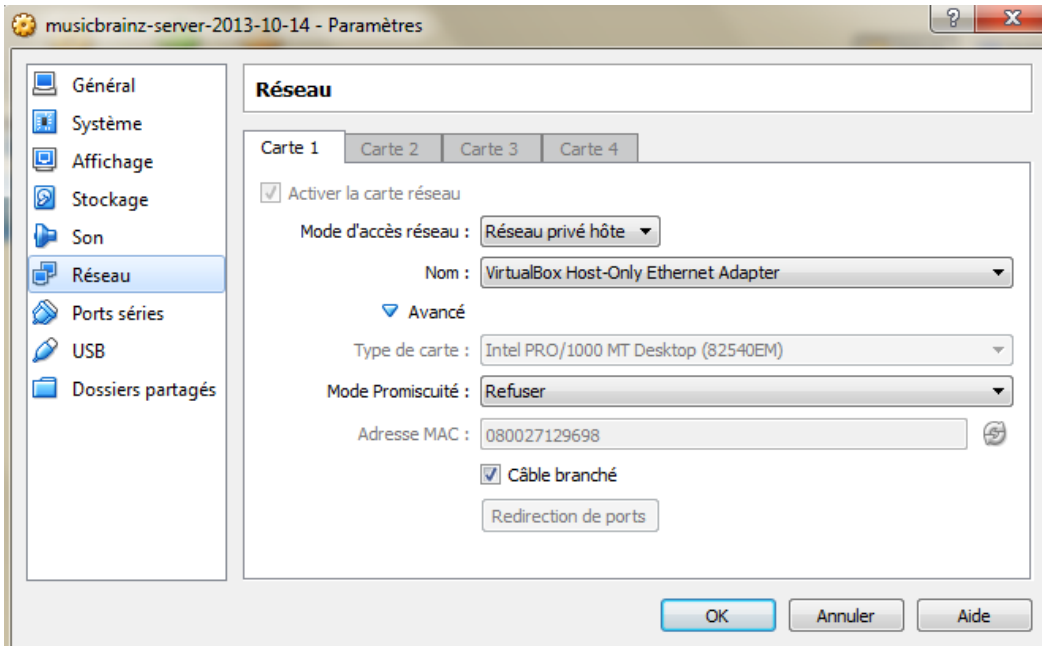
Nous avons travaillé sur le projet MusicBrainz. Le but était d'afficher les titres qui avaient sucité le plus de reprises. Pour celà nous avons utilisé la base de données fournie par MusicBrainz sous forme de machine virtuelle sous VirtualBox couplée à un serveur nodeJs qui effectue les requêtes.

Source de données (serveur MusicBrainz)

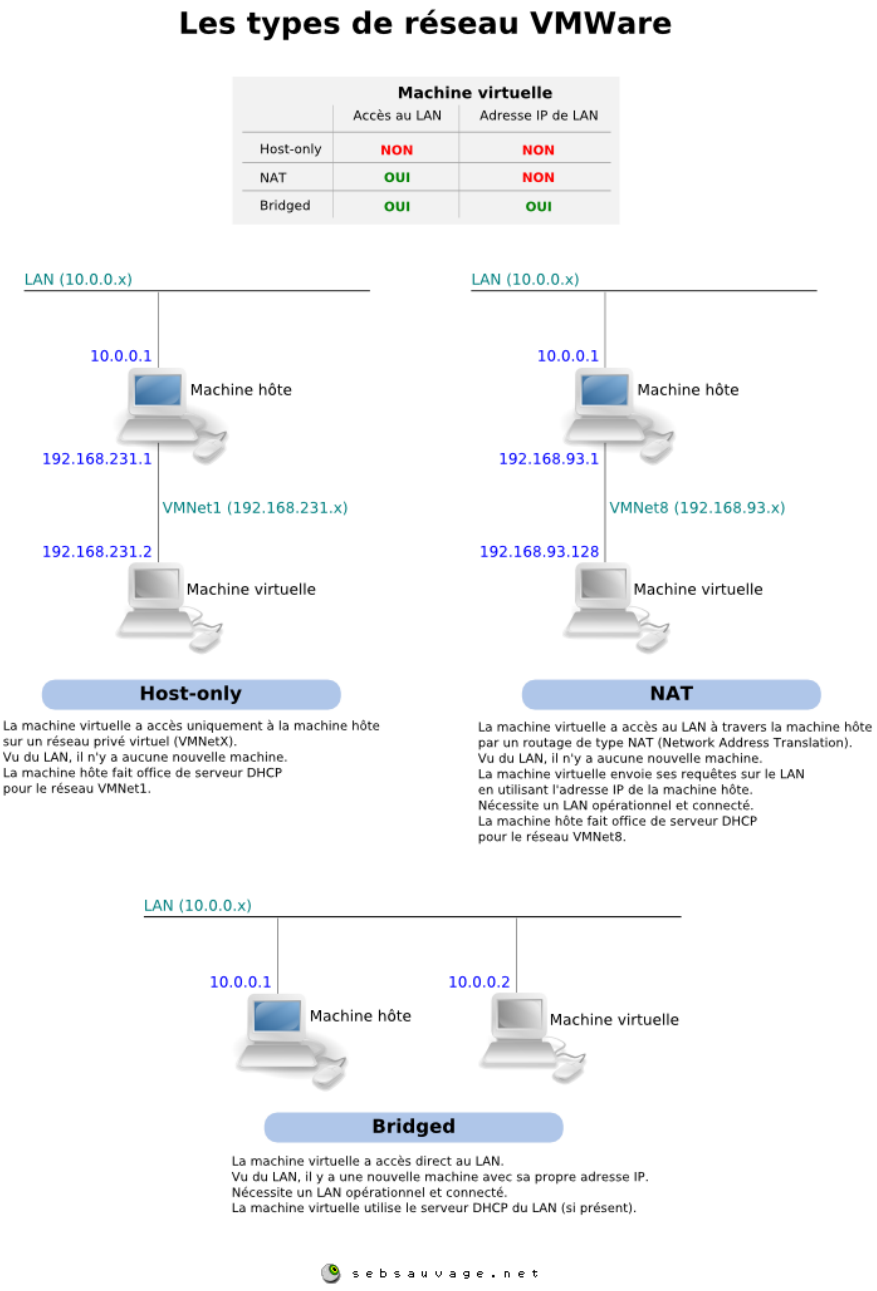
La source de données provient du site MusicBrainz. Nous avons téléchargé et configuré la machine virtuelle fournie.

Configurer le réseau

À notre domicile nous avions configuré le réseau en accès par pont, la machine avait donc sa propre adresse IP. Seulement le réseau de la faculté n'autorise pas une machine à avoir deux adresses IP. Nous n'obtenions donc plus d'adresse IPv4 pour notre machine virtuelle. Nous avons donc configuré le réseau en local uniquement.



Voici un schéma qui explique les différences entre les différentes configurations de réseau, les types de réseau VMWare sont les mêmes que ceux de VirtualBox.



Configurer les accès à la base de données

Activer l'authentification du client au serveur PostgreSQL

L'authentification permet au serveur de la base de données d'identifier quel client est autorisé à se connecter à la base de données.

On édite les configurations de PostgreSql :

```
sudo nano /etc/postgresql/9.1/main/pg_hba.conf
```

hba signifie : « host-based authentication » c'est-à-dire authentification fondée sur l'hôte. Chaque ligne est un enregistrement d'un type de connexion, une plage d'adresses IP (si approprié au type de connexion), un nom de base de données, un nom d'utilisateur, la méthode d'authentification à utiliser pour les connexions correspondant à ces paramètres. Nous nous servirons du format d'enregistrement suivant : host database user address auth-method

On ajoute la ligne :

```
host all all 192.168.56.101/24 trust
```

Le type de connexion host permet d'intercepter les connexions TCP/IP. La méthode d'identification trust autorise la connexion sans condition. On autorise ainsi tous les utilisateurs de la machine ayant l'adresse IP 192.168.56.101 (notre machine virtuelle) à se connecter à toutes les bases de données.

Autoriser les connexions en TCP/IP

On édite le fichier de configuration de PostgreSQL :

```
sudo nano /etc/postgresql/9.1/main/postgresql.conf
```

On ajouter la ligne :

```
listen_addresses='*'
```

Le paramètre 'listen_addresses' indique les adresses TCP/IP sur lesquelles écoute le serveur les connexions clientes. On donne accès à toutes adresses IP à la base de données.

Redémarrer le serveur PostgreSQL

```
sudo /etc/init.d/postgresql restart
```

Il faut toujours sauvegarder la machine virtuelle sinon les modifications apportées au fichier /etc/postgresql/9.1/main/postgresql.conf sont écrasées à chaque démarrage de la machine virtuelle.

Forcer l'adresse IP

On modifie le fichier /etc/network/interfaces pour forcer l'adresse IP de la machine virtuelle sinon à chaque utilisation de la machine l'adresse IP risque de changer.

On remplace :

```
iface eth0 inet dhcp
```

par :

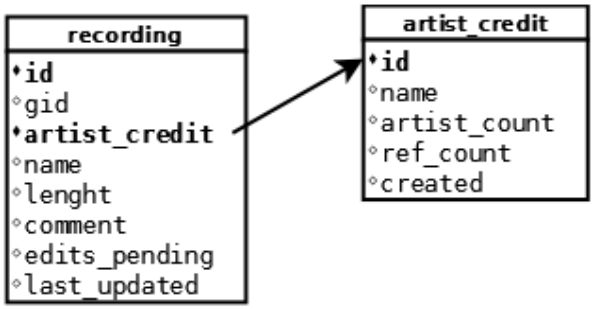
```
iface eth0 inet static
       address 192.168.56.101
```

La base

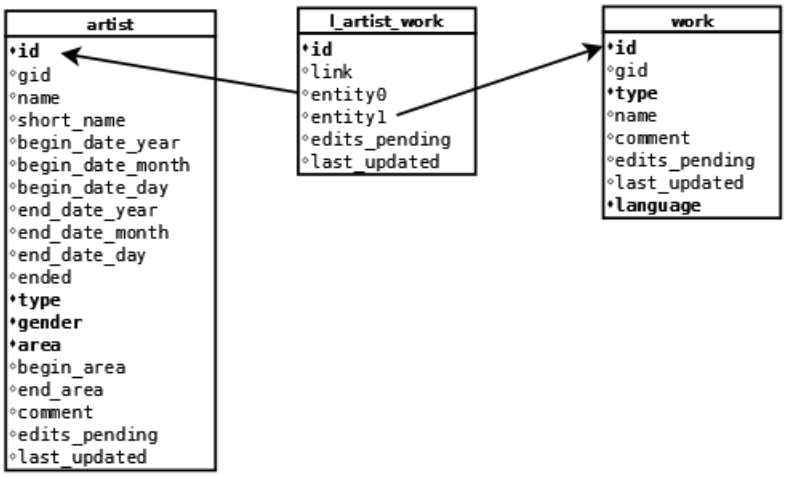
Dans la base de données MusicBrainz qui porte au total 195 tables (la commande \dt de PostgreSQL nous renseigne cette information), nous n'utiliserons que six tables dont deux de liaisons pour notre requête qui consiste à trouver les personnes ou groupes de musique qui ont repris une chanson par rapport à son titre. Puisqu'il peut exister deux chansons qui portent le même nom, nous les différencierons grâce à l'artiste qui a écrit cette chanson.

Nous allons dans un premier temps chercher les chansons qui portent le titre souhaité dans la table 'work'. Un 'work' est une création intellectuelle unique, en ce sens il n'est pas forcément musical (même si c'est majoritairement le cas). Un work peut avoir plusieurs enregistrements, 'recording', ce seront nos reprises. On trouvera donc nos reprises grâce à la table de liaison 'l_recording_work' où entity0 correspond au recording (reprise) et entity1 correspond au work (originale).

Nous allons ensuite chercher le nom des artistes qui ont joués les reprises via l'attribut 'artist_credit' et la table du même nom.



Pour trouver les artistes qui ont écrit les chansons originales 'work' nous irons chercher dans la table 'artist' grâce à la table de liaison 'l_artist_work' où l'attribut 'entity0' correspond à l'artiste et 'entity1' correspond à la chanson.



La requête est écrite en SQL. (cf. annexe A)

Serveur Node

Le serveur a été développé en nodeJs. C'est lui qui intercepte les requêtes de l'utilisateur et lui renvoie ensuite la page HTML correspondant avec les résultats.

Modules

Express C'est un mini-framework qui va permettre de gérer plus facilement les routes (url) de notre site.

Ejs Ce module nous sert de moteur de template afin d'écrire notre code de la vue directement dans un fichier dédié au lieu de l'écrire dans la réponse du serveur.

Postgres Grâce à ce module, il est possible d'effectuer une requête sur le SGBD PostgreSQL.

path On spécifie le dossier contenant les ressources statiques (css, img, js) sur le serveur.

queryString On récupère les paramètres passés dans l'URL.

http Obligatoire afin d'éxecuter des requêtes au serveur. Sans lui, on ne pourrait interroger le serveur et il ne pourrait encore moins nous répondre.

Couplage à la machine virtuelle

Il est impératif de spécifier le nom d'hôte de la machine virtuelle afin d'y effectuer les requêtes.
L'adresse IP correspond à celle que nous avons renseigné lors de la configuration du serveur MusicBrainz.

```
var conString = "postgres://musicbrainz@192.168.56.101:5432/musicbrainz"
    ;
```

Auto-complétion

Du fait que l'on recherche les titres complets en base de données, nous avons choisi d'implémenter un système d'auto-complétion de titre de chanson dans la barre de recherche.

Nous avons pour cela utilisé jQuery et jQuery UI et son plugin autocomplete. Nous récupérons d'abord les caractères saisis par l'utilisateur dans le champs de recherche. Ensuite, le plugin effectue une requête ajax à l'adresse "http ://-musicbrainz.org/ws/2/work/ ?query=" en rajoutant les caractères récupérés du formulaire.
Cette adresse, "http ://musicbrainz.org/ws/2/work/ ?query=chaine+de+caracteres" nous renvoie vers un fichier xml en rapport avec "chaine de caractère".

On parse ensuite ce fichier et l'on récupère la valeur de la balise <work> que l'on affiche dans l'autocomplete du formulaire.

Le plugin autocomplete que nous avons utilisé peut aussi agir directement sur un fichier XML. Nous avons donc essayé de générer un fichier depuis la base de données avec la requête :

```
SELECT xmlforest(name) FROM work;
```

Cette requête nous renvoie du XML du type :

```
<name>Titre de la chanson</name>
```

Cependant, en JavaScript, nous n'avons pas réussi à écrire notre résultat dans un fichier. C'est pourquoi nous avons opté pour la solution précédente (cf fonction makeXML() dans le script serveur).

Difficultés rencontrées et améliorations

Lors du développement de ce projet, nous nous sommes retrouvés confronté à certaines difficultés notamment liées à la requête SQL et à la base de données.

Base de données

Une des premières difficultés que nous avons rencontrée est de bien comprendre la base de données, plusieurs tables ne sont pas décrites dans le schéma, les tables de liaison par exemple. Il nous a fallu tester plusieurs requêtes sur différentes tables avant de trouver la bonne. La multitude de tables nous pose alors un autre problème : le temps d'exécution de la requête.

Optimisation de la requête

La requête est très longue, nous avons donc essayé de l'optimiser sans véritable succès. La fonction EXPLAIN ma_requête permet de voir le plan de la requête (coût, utilisation d'index...)

Indexation des tables

Nous avons tenté d'utiliser des index B-tree, le type d'index utilisé par défaut. D'après la documentation, ces index permettent de "traiter les égalités et les recherches sur des tranches de valeurs des données qui peuvent être triées". La requête s'effectue normalement plus rapidement. Nous avons créé des index sur champs des tables que nous utilisions, mais cette opération n'a pas eu l'effet escompté.

Requête pour créer des index :

```
CREATE INDEX name_work_idx ON work (upper(name));
```

Partitionnement

Le partitionnement permet de partitionner la base de données pour que celles qui ne sont pas utilisées fréquemment prennent moins de ressources. Cela permet aussi de diviser une table. En PostgreSQL, le partitionnement s'effectue par héritage. Nous voulions donc partionner la table "work" en table "work_premierLettreDuName". Chaque classe fille hérite des champs de la table mère. Ainsi selon la première lettre du titre demandé, nous allions chercher l'id dans la table fille correspondante et nom dans la classe mère très conséquente. Cette amélioration n'a pas eu l'effet attendu, voir même l'effet inverse. Nous sommes donc revenu à notre requête de départ.

Création d'une table fille :

```
CREATE TABLE work_a(
        #nouveau champs en plus de ceux de la classe parent, si besoin
)INHERITS (work);
```

Copie de la donnée de la classe mère vers la classe fille :

```
SELECT * FROM work_s (
        SELECT * FROM work
        WHERE name LIKE 'S%'
);
```

Il faut ensuite supprimer la donnée de la table mère pour ne pas avoir de doublon.

Conclusion

D'après ce que nous avons pu voir, ce serait les jointures entre plusieurs tables qui sont longues à l'exécution. Apparemment c'est un problème courant qui est assez difficile à résoudre. Nous pourrions limiter l'utilisation de jointures dans notre requête et réaliser plusieurs requêtes plus simples. Il faudrait alors récupérer le résultat de chaque requête et les sauvegardées dans notre tableau JavaScript. Nous n'avons pas mis en pratique cette théorie. On pourrait peut-être aussi gérer l'affichage de la données différemment : ne pas attendre que la ou les requête(s) ait/aient fini de s'exécuter mais afficher quelques résultats au fur et à mesure de l'exécution.

Auto-complétion

Comme vu précédemment (voir chapitre Auto-complétion), nous utilisons le site de MusicBrainz pour gérer l'auto-complétion dans notre barre de recherche. Cette utilisation a cependant une limite : nous sommes obligé de rentrer un

mot complet et faire un espace avant d'avoir accès à l'auto-complétion. Le site de MusicBrainz ne génère pas de fichier XML dans d'autres conditions. Une des améliorations possible serait donc d'avoir ses propres fichiers xml afin de proposer une auto-complétion plus précise, au bout de trois lettres tapées. Si le fichier XML est trop lourd, nous pourrions le décomposer en plusieurs fichiers XML par rapport au trois premières lettres.

Conclusion

Ce projet nous a permis de lier la virtual box à notre serveur nodeJS afin d'afficher la liste des reprises. Malgré une configuration de la machine virtuelle un peu laborieuse (clavier en qwerty, perte des données de config lorsque l'on éteint, ...) l'appli est fonctionnelle et nous retourne une liste des artistes ayant repris une chanson.

Nous avons découvert un nouveau SGBD [1] qui fonctionne pratiquement de la même manière que MySQL. Dans ces deux SGBD les requêtes sont écrites en SQL ce qui a permis une adaptation très rapide.

1. Système de Gestion de Bases de Données

Annexe A

Requête SQL

```
'WITH all_works AS ( ' +
                'SELECT * ' +
                'FROM work ' +
                'WHERE UPPER(name) LIKE UPPER(\'' + params['titre'] + '\')
                    ' +
                ')'+
        ', all_recordings_works AS ( ' +
                'SELECT * ' +
                'FROM l_recording_work ' +
                'WHERE entity1 IN (SELECT id FROM all_works) ) '+
        ', all_recordings AS ( '+
                'SELECT * '+
                'FROM recording '+
                'WHERE id IN (SELECT entity0 FROM all_recordings_works)
                    ) '+
'SELECT DISTINCT (artist_credit.name, artist.name, l_recording_work.
    entity1) '+
'FROM artist_credit INNER JOIN all_recordings ON all_recordings.
    artist_credit
= artist_credit.id '+
'INNER JOIN l_recording_work ON l_recording_work.entity0 =
    all_recordings.id ' +
'INNER JOIN l_artist_work ON l_recording_work.entity1 =
    l_artist_work.entity1 '+
'INNER JOIN artist ON artist.id = l_artist_work.entity0 '+
'WHERE artist_credit.id IN (SELECT artist_credit FROM all_recordings)
```

Cette requête, à première vue complexe, comporte en réalité quatre requêtes imbriquées.

1. recherche l'id de la chanson dans la table "work" par rapport au titre

2. trouve les reprises par rapport aux chansons originales
3. trouve l'id de l'artiste pour la reprise
4. trouve le nom de l'artiste des reprises et l'écrivain pour les originales

Annexe B

Base de données complète