Table des matières

SERVEUR 2

Choix Technique 2

Installation du serveur: 2

Mise en route 3

Backup du serveur 3

La base de donnée mySql 4

Connexion 4

Ajouts des données 4

Remplissage de la base de donnée: 5

CreateDatabaseTable.js 6

StoreData.js 6

UpdateData.js 7

Les appels au serveur 7

Pour l'application android 8

Pour les tests 8

Pour le site web 8

Format de donnée requêtes GET 8

Les requêtes POST 9

Requêtes 148.60.11.208:3000/android/data/getroutes 9

Base de donnée Neo4j 10

Connexion à la base Neo4j 10

Création des noeuds 10

Création des relations 10

Mise à jour des tables SQL 11

Visualisation du graph 11

Procédure d’installation. 11

Pré-requis 11

Création de la base de données Mysql 11

Création du graph Neo4j 13

Mise à jour des données. 13

# SERVEUR

# Choix Technique

Pour ce projet, nous avons choisi pour le serveur d'utiliser NodeJS. Nous avons fait ce choix afin de pouvoir gérer de façon asynchrone les requêtes reçues par le serveur. Nous utilisons la dernière version stable de NodeJs qui est la version:

node v0.8.15

Pour stocker les informations statique comme les horaires de bus, les arrêts de vélo ainsi que les gares, nous avons fait le choix d'une base de donnée MySql.

Nous utilisons la version suivante de MySql:

MySql v5.1

Pour ce qui est du graphe, nous avons choisi d'utiliser une base de donnée de graphe. Après recherche, nous avons décidés d'utiliser:

Neo4j

C'est une base de donnée noSql, le module nodeJS "node-neo4j" permet de pouvoir interagir depuis le serveur nodeJs vers la base de donnée neoJs.

# Installation du serveur:

Le serveur utilisé est un serveur NodeJs. Cela veut dire qu'il est codé en JavaScript. Pour gérer notre application, nous avons choisi d'utiliser le Framework express. Pour pouvoir installer différents paquets sur le serveur, NodeJs possède un manager de paquets appelé npm.

Depuis nom, nous pouvons installer le Framework express de la façon suivante:

> sudo npm install express -g

Une fois que le Framework est installé, nous pouvons alors crée une application de la façon suivante:

> express --sessions --css stylus --ejs <App name>

L'exécution de cette commande va créer un dossier portant le nom de l'application. Dans ce dossier, nous trouverons les fichiers et dossiers suivants:

|- node-modules/

|- public/

|- stylesheets/

|- javascript/

|- routes/

|- views/

|- app.js

Le fichier app.js est le fichier dans lequel se trouvera tout le code contenant le comportement du serveur.

Pour lancer le serveur il suffit d'exécuter la commande suivante:

> node app.js

# Mise en route

Afin de rendre tous les services du serveur disponible, il faut le démarrer. Pour cela, on peut exécuter la commande:

> node app.js

Cela va démarrer le serveur. Cependant nous n'avons aucun contrôle sur son arrêt ou son démarrage. Afin de palier à ce problème, nous utilisons le modules "forever" (https://github.com/nodejitsu/forever).

L'installation se fait de la façon suivante:

> sudo npm install forever

Une fois ce module installé, nous pouvons démarrer, arrêter, ou redémarrer le serveur de la façon suivante:

> forever start app.js

> forever stop app.js

> forever restart app.js

# Backup du serveur

En cas de problème sur le serveur, nous avons écrit un script bash qui permet de sauvegarder la totalité du serveur. En cas de problème, nous pourrons utiliser cette sauvegarde afin pour rétablir le serveur dans l'état dans lequel il était au moment de la sauvegarde.

Le script se trouve dans le dossier suivant:

|- Projet-BMA

|- BmaServeur

|- Android-prod

|- SiteWeb

|- serverBackup

|- backupScript.sh

Pour exécuter le script, il suffit de d'entrer la commande suivante:

> sh backupScript.sh

# La base de donnée mySql

## Connexion

Afin de pouvoir interagir avec les données, il est possible de se connecter à celle-ci en ligne de commande. Pour cela:

> mysql -u "username" -p

En cas de problème, nous pouvons arrêter, démarrer ou redémarrer le serveur mysql de la façon suivante:

> service mysql stop

> service mysql start

> service mysql restart

Une fois la base de donnée remplie, pour des raisons de sécurité, ou pour prévenir tous problème sur le serveur, nous pouvons faire une backup de la base de donnée de la façon suivante:

> mysqldump -u root -p test > alldb\_backup.sql

Et nous pouvons ainsi la restorer de la façon suivante:

> mysqldump -u root -p test < alldb\_backup.sql

## Ajouts des données

Afin de sauvegarder toutes les informations concernant les horaires de bus, train ... et les arrêts, toutes les données sont sauvegardé dans une base de données mySql.

La connexion entre le serveur et la base de données s'effectue grâce au module mode-mysql (https://github.com/felixge/node-mysql).

L'installation de ce module s'effectue de la façon suivante:

Tout d'abord ce module:

> sudo npm install mysql@2.0.0-alpha5

Puis:

> sudo npm install git://github.com/felixge/node-mysql.git

Une fois ce module installé, nous pouvons nous connecté de la façon suivante à la base de donnée:

var mysql = require('mysql');

var connection = mysql.createConnection({

host : 'localhost',

user : 'me',

password : 'secret',

});

connection.connect();

Puis les queries se font de la façon suivante:

connection.query('SELECT \* FROM Tables', function(err, result){

// Traitement du resultat de la query

});

## Remplissage de la base de donnée:

Pour ajouter des données à notre base, nous avons écrit des scripts (en JavaScript) qui permettent de parsé les différentes données présentes dans les fichiers fournis par la STAR et la SNCF ainsi que les données récupérées grâce aux appels Ajax vers l'API de la STAR (données vélo et métro).

Les scripts réalisant ces actions ce trouve dans le dossier:

|-One\_Time\_Sql/

|- CreateDatabaseTable.js

|- RemoveDataFromSqlDb.js

|- StoreBikeStops.js

|- StoreBusRoutes.js

|- StoreBusStop\_Times.js

|- StoreBusStops.js

|- StoreBusTrips.js

|- StoreMetroStops.js

|- StoreBorneElec.js

Pour exécuter l'un de ces scripts, il suffit de ce placé dans le dossier "One\_Time\_Sql" puis d'exécuter la commande suivante:

> node <Nom du Script a éxecuter>

### CreateDatabaseTable.js

Ce fichier créé les différentes tables de notre base de donnée. Pour créer une table, il suffit d'exécuter l'équivalant d'une requête Sql en JavaScript. C'est à dire transformer cette requête Sql:

// Création d'une table Sql

CREATE TABLE TerStops (

Stop\_id int,

Stop\_name VARCHAR(100),

Stop\_lat VARCHAR(100),

Stop\_lon VARCHAR(100),

PRIMARY KEY(Stop\_id));

En JavaScript:

// Création d'une table Sql en javascript

connection.query('CREATE TABLE TerStops ( ' +

'Stop\_id int,' +

'Stop\_name VARCHAR(100),' +

'Stop\_lat VARCHAR(100),' +

'Stop\_lon VARCHAR(100),' +

'PRIMARY KEY(Stop\_id))',

function(err, result){

// Case there is an error during the creation

if(err) {

console.log(err);

} else {

console.log("Table Ter\_Stops Created");

}

});

L'exécution de ce fichier nous permet donc de créer toutes les tables de notre base de donnée et ainsi de créer un schéma de donnée comme celui ci:

### StoreData.js

Afin d'ajouter données dans notre base de donnée, nous avons créé des scripts qui lors de leur exécution vont faire un appel Ajax vers les différents serveurs, ou alors parser les fichiers contenants les données.

Ajouter des données dans une table se fait de la façon suivante en Sql:

INSERT INTO <Nom de la Table> SET <Colonne> = <Valeur>, <Colonne> = <Valeur>;

Ce qui donne en JavaScript:

connection.query('INSERT INTO <Nom de la table> SET <Colonne> = <Valeur>, <Colonne> = <Valeur>',

function(err, result){

// Gestion du resultat.

});

Note: Afin de pouvoir charger des fichiers dans un script, il faut procéder de la façon suivante:

fs.readFile(<Location du fichier>, function (err, data) {

// Case there is an error

if (err) {

console.log("File not load");

throw err;

}

// Traitement des données

});

Note 2: Le fichier Stop\_Times contenant beaucoup de donnée, nous sommes obligé de le parcourir en deux fois.

Pour cela, nous avons créés deux fonctions, une qui va ajouter la première moitié des données dans la base et une seconde qui une fois l'exécution de la première fonction terminer va ajouter le reste des données dans la base.

Nous somme obligé de procéder de cette façon, sinon nous obtenons l'erreur:

Fatal Error: Process out of memory

### UpdateData.js

Afin de donner plus de valeur aux données, nous avons écrit un script qui ajoute à la table des arrêts de bus, les numéros de lignes de bus qui s'y arrête. Pour cela nous avons juste effectué un JOIN en sql.

# Les appels au serveur

Les appels faits au serveur depuis l'application sont des appels HTTP. Le traitement de chaque appel depend de l'url que la requête appel. Les urls sont définis de la façon suivante:

## Pour l'application android

148.60.11.208:3000/android/data/bike

148.60.11.208:3000/android/data/bus

148.60.11.208:3000/android/data/train

148.60.11.208:3000/android/data/borneelec

148.60.11.208:3000/android/data/getroutes // Necessite paramètre

148.60.11.208:3000/android/nextdeparture // Necessite le paramètre stopId correspondant à un arrêt de bus

## Pour les tests

148.60.11.208:3000/test/bike

148.60.11.208:3000/test/bus

148.60.11.208:3000/test/metro

148.60.11.208:3000/test/borneelec

148.60.11.208:3000/test/testgraphe

## Pour le site web

148.60.11.208:3000

Les appels a ces Url ne nécessite pas d'envoi de paramètre de la part de l'application, cela veut dire que si l'application appel une de ces trois Url, elle aura pour résultat les données concernant soit les bus, les vélos ou les trains.

## Format de donnée requêtes GET

Les données envoyées par le serveur sont de la forme suivante:

[{ Stop\_id: '"4604"',

Stop\_name: '"Pommiers"',

Stop\_lat: '"48.21008788"',

Stop\_lon: '"-1.62587418"',

Stop\_code: '"4604"',

Stop\_desc: '"Chevaigné"',

Zone\_id: '""',

Stop\_url: '""',

Location\_type: '""',

Parent\_station: '""',

Stop\_timezone: '""',

Wheelchair\_boarding: '"1"',

Line\_short\_name: null,

Line\_long\_name: null },

{ Stop\_id: '"4605"',

Stop\_name: '"Fonderies"',

Stop\_lat: '"48.21442422"',

Stop\_lon: '"-1.62878295"',

Stop\_code: '"4605"',

Stop\_desc: '"Chevaigné"',

Zone\_id: '""',

Stop\_url: '""',

Location\_type: '""',

Parent\_station: '""',

Stop\_timezone: '""',

Wheelchair\_boarding: '"1"',

Line\_short\_name: null,

Line\_long\_name: null },

... ]

## Les requêtes POST

Afin d'obtenir des résultats plus précis ou ne concernant qu'un seul arrêt ou une location spécifique,

l'application peut faire une requête vers le serveur en lui transmettant des paramètres.

### Requêtes 148.60.11.208:3000/android/data/getroutes

Cette requête nécessite l'envoie de paramètres de la part de l'utilisateur.

Ils devront etre mis sous cette forme:

depart = "type:id"

arrive = "type:id"

bus = true/false

bike = true/false

metro = true/false

Les variables contenant les informations sur le départ et l'arrivée seront constitués de deux données, le type qui pourra être Bus ou Bike ou Metro suivi de l'Id de l'arrêt sélectionnée.

Les trois autres paramètres concerne les chois de l'utilisateur pour savoir si il préfère utilisé le bus, le vélo ou le métro.

# Base de donnée Neo4j

Cette base de donnée contient le graphe de tous les trajets possibles.

## Connexion à la base Neo4j

Afin de ce pouvoir interagir depuis le serveur nodeJS vers la base de donnée neo4j, nous avons besoin d'installer le module neo4js (https://github.com/neo4j/neo4js). Pour cela nous exécutons la commande suivante:

> sudo npm install node-neo4j

Tous les fichiers relatifs à la base de donnée Neo4j se trouvent dans le dossier:

|- OneTime\_Neo4j/

|- CreateGraph.js

Démarrer le serveur noe4j, il faut ce placer dans le dossier Neo4j puis exécuter la commande:

> bin/neo4j start

## Création des noeuds

Pour créer notre graphe, nous devons d'abord créer tous les noeuds qui représentent chaque arrêt de bus, vélo, métro ...

Pour créer un noeud, nous faisons une requête à la base de donnée puis nous créons le noeud de la manière suivante:

var noeud = graph.insertNode({name: <Nom du noeud>, latitude: ..., longitude: ...},

function(err, result){

// Callback function

});

## Création des relations

Une fois les noeuds du graph créés, il faut créer les relations qui lie chaque noeud entre-eux. Notre comprend plusieurs types de noeuds et de relations.

Noeuds: Arrêts de Bus

Arrêts de Vélos

Arrêts de Métro

Arrêts de Train

Relations: Distance & temps à vélo entre chaque arrêt de vélo

Distance & temps en bus en bus entre chaque arrêt de bus (en fonction des horaires)

Distance & temps en Metro entre chaque arrêt de métro

Distance & temps à pied entre chaque arrêt de bus

Distance & temps en Train entre chaque arrêt de train (en fonction des horaires)

Distance & temps à pied entre arrêts de métro et arrêts de bus

Distance & temps à pied entre arrêts de métro et arrêts de vélo

Distance & temps à pied entre arrêts de métro et arrêts de train

Distance & temps à pied entre arrêts de bus et arrêts de vélo

Distance & temps à pied entre arrêts de train et arrêts de bus

Distance & temps à pied entre arrêts de train et arrêts de vélo

## Mise à jour des tables SQL

Une fois le graphe créé, chaque noeud équivaut à un arrêt de bus ou métro ou vélo ou train. Dans le graph, chaque noeud possède un ID propre à lui. Afin de pouvoir accéder facilement au noeuds souhaiter, nous ajoutons aux différentes tables comprenant les arrêts de bus, vélos et métro la colonne suivante:

NodeId

Dans cette colonne se trouve l'id de l'arrêt dans le graphe.

## Visualisation du graph

Neo4j possède une console.

La commande suivante permet d'obtenir une vue graphique du graphe:

start n=node(\*)

match n-[r?]->m

return n,type(r),m

# Procédure d’installation.

## Prérequis

Avant de pouvoir lancer le serveur, il faut que NodeJs soit installé sur le serveur ainsi que Mysql et Neo4j.

## Création de la base de données Mysql

Une fois tout installé, il faut crée la base de donnée. Pour cela nous avons créés des scripts qui s’en chargent.

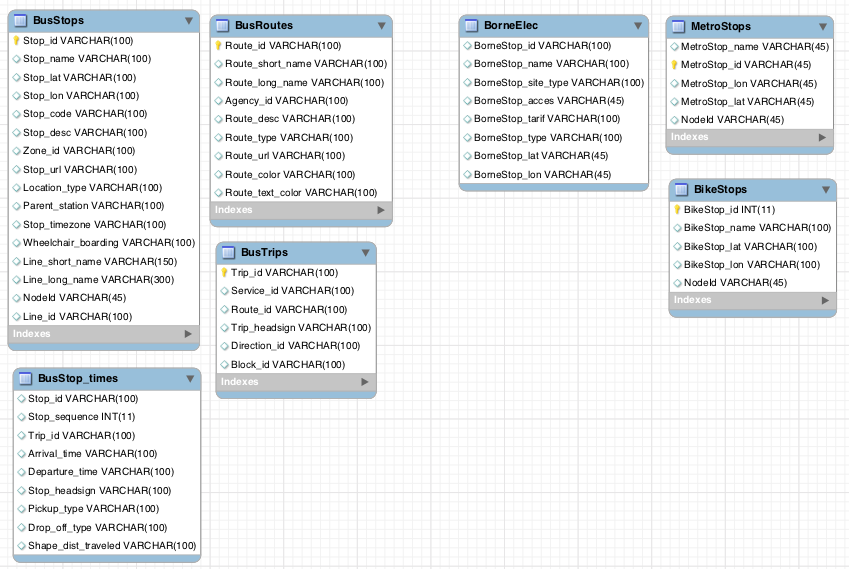
Tout d’abord il faut lancer le script qui va créer toutes les tables.

Pour cela il faut exécuter la commande

* node CreateDatabaseTable.js

Ce fichier se trouve dans le dossier One\_time\_sql/DatabaseCreation.

Voici une représentation des différentes tables de notre base de données.



En ce qui concerne l’ajout des données dans la base, il faut exécuter chaque script qui ce trouve dans les dossiers :

* Star\_Script
* BorneElec\_Script
* Ter\_Script

Dans le dossier Star\_Script se trouve tous les scripts qui ajoutent les données concernant les services de la STAR. C’est à dire les données concernant les vélos, les métros, et les bus.

Le dossier BorneElec\_Script comprend l’ajout des données des bornes électriques.

Et enfin le dossier Ter\_Script contient le script ajoutant les arrêts de train dans les tables de la base de données.

## Création du graph Neo4j

Une fois la base de données MySql créée, nous pouvons créer le graphe.

Comme pour la base de donnée MySql, nous avons écrit des scripts qui vont de charger de la création du graphe.

Ces scripts se trouvent dans le dossier One\_time\_neo4j. De la même façon que pour créer la base de donnée MySql, il faut exécuter chacun des scripts.

Les scripts qui se trouvent dans le dossier NodeCreation vont créer tous les nœuds du graphe. Il faudra donc tous les exécuter en premier.

Une fois ces scripts exécuter, nous allons pouvoir créer les relations entre chaque nœuds du graphe. Cela se fait en exécutant les tous les scripts qui se trouvent dans le dossier RelationCreation.

# Mise à jour des données.

En cas de changement des données de la STAR, il faut remettre la base de donnée à jour.

Tout d’abord il faut supprimer les anciennes données. Cela se fait facilement grâce au script qui se trouve dans le dossier One\_time\_sql/DatabaseCreation.

Pour supprimer toutes les données, il suffit d’exécuter le script RemoveDataFromSqlDb de la façon suivante:

Node RemoveDataFromSqlDb.js

Une fois cela fait, il faut remplacer les fichiers de la STAR (ou autre) qui se trouvent dans le dossier gtfsData\_STAR. Une fois cela fait, il faut recréer la base de donnée MySql de la même façon que la première fois.

# Android Application

## Choix technique:

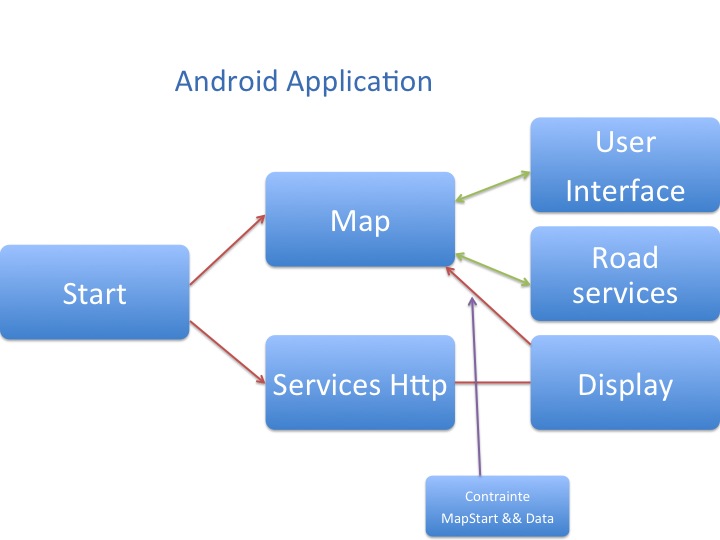
L’application est composée des outils suivants :

* OpenStreetMap(Osmdroid, slf4j)
* Json
* Intent Services

## Découpage des fichiers de l'application:

Il y a un fichier de chargement, Firstview instancie un service (HttpRequestClass).

Et une MapActivity (MapViewClass) qui intégrer les problèmes de GPS (LocationListener) et qui sont typé génétiquement avec des overlay qui permet la mise en place des informations sur la carte.



## Gestion des appeles serveur:

### Service HttpRequestClass appel serveur :

il a pour but la récupération des données de manière asynchrone et leurs mises a disposition pour pour qu'elles puissent être afficher sur la map.

Pour cela il y un appel Http GET au serveur aux différentes adresses défini dans la partie serveur.

### Cette fonction a une double utilisation :

la gestion des appels directe aux différents serveurs d'information.

la gestion des caches sur les différentes informations pour évite de surcharger les serveurs

Puis cette information est broadcasté pour que le reciver la traiter et lorsque la map est prête les informations puissent être affiché pour l'utilisateur.

## Gestion de la Map:

Création d'une map OSMDroid

écriture des différents arrête selon les choix de l'utilisateur

Gestion des choix utilisateur

Gestion des paramétrages utilisateur

## Gestion d'itinéraire

Encore une fois un appelle à un service avec récupération de l'ID des stations grâce à la liste mise en place l'or du premier appel.

Puis appel HTTP POST

Puis utilisation des données et de l'outil Road.

Enfin, demande d'affichage pour l'utilisateur.

## Road

nous utiliserons pour cette partie L'osmDroid bonus pack dont la documentation est disponible ici :

http://code.google.com/p/osmbonuspack/wiki/Tutorial\_1

Pour utiliser cet outil nous devons mettre en place une classe asynchrone qui ce nome Road display qui permet a l'outil get road de faire dans appel HTTP qui n’est pas autorisé pas le thread principal.