

CFPT ÉCOLE D'INFORMATIQUE

TECHNICIEN ES EN INFORMATIQUE

TRAVAIL DE SEMESTRE

---

# TPGraph

DOCUMENTATION

---

*Eleve:*

Dimitri LIZZI

*Enseignante:*

Mme TERRIER

17 novembre 2015

# Table des matières

<b>Introduction</b>	<b>4</b>
<b>Cahier des charges</b>	<b>5</b>
Sujet . . . . .	5
But . . . . .	5
Spécifications . . . . .	5
Restrictions . . . . .	5
Environnement . . . . .	6
Livrables . . . . .	6
Redditions . . . . .	6
Planification . . . . .	6
<b>Analyse de l'existant</b>	<b>8</b>
Site des TPG . . . . .	8
Öffi . . . . .	8
Itinera . . . . .	11
Site des CFF . . . . .	11
Conclusion . . . . .	11
<b>Analyse fonctionnelle</b>	<b>13</b>
Esquisses de l'interface . . . . .	13
Page d'accueil . . . . .	13
Page d'affichage des résultats de recherche d'itinéraire . . . . .	13
Base de données orientée graphe . . . . .	15
Neo4j . . . . .	16
Framework web Django . . . . .	16
Docker . . . . .	17

<b>Analyse organique</b>	<b>18</b>
Structure des containers Docker . . . . .	18
Structure du graphe de l'application . . . . .	18
commercial_stop . . . . .	19
physical_stop . . . . .	20
line . . . . .	21
route_step . . . . .	22
destination . . . . .	22
Requête d'itinéraires dans le graphe . . . . .	22
Application Django . . . . .	25
Modèle . . . . .	25
Views . . . . .	25
Routage . . . . .	28
Tâches <i>Celery</i> . . . . .	28
<b>Guide de maintenance</b>	<b>29</b>
Installation de l'application en local . . . . .	29
Mise à jour de l'application après modification des sources . . . . .	29
Utilisation de l'application . . . . .	30
Accès à l'interface web . . . . .	30
Description de la page d'accueil . . . . .	30
Chargement des données . . . . .	31
Recherche de routes . . . . .	31
Interfaces secondaires . . . . .	31
Monitoring des tâches de fond : <i>Flower</i> . . . . .	32
Interfaces d'administration de Neo4J . . . . .	32
<b>Protocole de test</b>	<b>33</b>
<b>Conclusion</b>	<b>34</b>
Retour sur la planification . . . . .	34
Retour sur les technologies . . . . .	34
Docker . . . . .	34
Django . . . . .	34

Neo4j . . . . .	35
Retour sur l'application . . . . .	35
Intérêt personnel . . . . .	35

# Introduction

Dans le cadre du premier travail de semestre de ma deuxième et dernière année de formation de Technicien ES en informatique, au *Centre de Formation Professionnelle Technique*, mon projet est une application de calcul d'itinéraires pour les *Transports Publics Genevois*.

Cette application se base sur les données en temps réel fournies par les TPG au travers de leur API <sup>1</sup> Open Data.

L'intérêt de ce travail est d'étudier les bases de données orientées graphes, qui sont réputées pour travailler de manière très performante avec ce type de données très relationnel, au contraire des bases de données relationnelles classique telles que MySQL, MariaDB, SQL Server ou encore Oracle.

---

1. *Application Programming Interface*, ou interface de programmation en Français.

# Cahier des charges

## Sujet

Calcul d'itinéraire pour les Transports Publics Genevois avec les données des horaires en temps réel.

## But

Application de calcul d'itinéraire des Transports Publics Genevois (TPG) en se basant sur les données en temps réel mises à disposition dans leur API "Open data".

Il est en effet possible, depuis le site web des TPG, de calculer des itinéraires, mais ces derniers se basent sur des horaires statiques, qui ne changent pas en fonction des perturbations du réseau. Depuis quelques années, les TPG ont rajouté sur leur site la possibilité de consulter des horaires en temps réel, qui changent selon la position des bus sur le réseau, et qui sont donc beaucoup plus précis. Ces données sont mises à disposition dans une API ouverte au grand public. Malheureusement, ces données indiquent uniquement les heures d'arrivée des transports aux différents arrêts mais ne permettent pas de calculer des itinéraires.

Le but de l'application est donc de récupérer ces données en temps réel et de les utiliser pour calculer des itinéraires (pour les quelques heures qui suivent uniquement) qui seraient plus précis que ceux fournis par le site, car se basant sur des horaires en temps réel, plutôt que sur des données statiques.

## Spécifications

L'application sera composée de plusieurs parties. Elle devra:

1. Récupérer les données des TPG périodiquement.
2. Enregistrer les données récupérées dans une base de données de type graphe.
3. Fournir une API permettant de lancer des calculs d'itinéraires via des requêtes HTTP.
4. Mettre à disposition de l'utilisateur une interface web permettant de rechercher facilement un itinéraire, en se basant sur l'API.

## Restrictions

- L'application pourra avoir quelques minutes de retard sur les horaires en temps réel, car il faudra prendre en compte le temps de téléchargement et de chargement des données dans la base.
- L'interface graphique (interface web) sera très minimaliste. Les efforts seront concentrés sur le back-end et non le front-end.

## Environnement

- Un serveur (PC compatible x86)
- Système d'exploitation: Ubuntu linux 15.04
- Navigateur web moderne: Firefox  $\geq 37$
- Base de données orientée graphe: Neo4j
- Langage de programmation: Python
- Framework web: Django

## Livrables

- Poster
- Code source et projet
- Documentation
- Présentation
- Journal de bord

## Redditions

Mardi 29 septembre: reddition du poster  
Mardi 17 novembre: reddition finale  
Mardi 24 novembre: présentations

## Planification

La planification du projet est décrite ci-dessous. Il faut noter que les dates indiquées sont celles des cours, tous les mardi, mais les tâches pourront être effectuées tout le long de la semaine, en plus des cours consacrés au projet.

- 25 août 2015
  - Présentation du cours
  - Rédaction du cahier des charges
- 1er septembre 2015:
  - Rédaction de la planification
  - Mise en page du cahier des charges
  - Recherches sur Neo4j et rédaction de l'analyse fonctionnelle liée
  - Installation de l'environnement de travail
- 8 septembre 2015:
  - Recherches sur Neo4j et rédaction de l'analyse fonctionnelle liée
  - Essais d'architecture de données
  - Recherches sur l'architecture de l'application, les librairies à utiliser, et rédaction de l'analyse fonctionnelle liée
- 15 septembre 2015:
  - Essais d'architecture de données

- Recherches sur l'architecture de l'application, les librairies à utiliser, et rédaction de l'analyse fonctionnelle liée
- Conceptualisation de l'API et rédaction de l'analyse fonctionnelle liée
- Conceptualisation de l'interface et rédaction de l'analyse fonctionnelle liée
- 22 septembre 2015:
  - Conceptualisation de l'API et rédaction de l'analyse fonctionnelle liée
  - Conceptualisation de l'interface et rédaction de l'analyse fonctionnelle liée
  - Rédaction du poster
- 29 septembre 2015:
  - Rédaction et finalisation du poster
  - Développement du chargement des données dans la base de données
  - **Reddition du poster**
- 6 octobre 2015:
  - Développement du chargement des données dans la base de données
  - Développement des méthodes de recherche d'itinéraire
  - Rédaction de l'analyse fonctionnelle
- 13 octobre 2015:
  - Finalisation de l'analyse fonctionnelle
  - Développement du chargement des données dans la base de données
  - Développement des méthodes de recherche d'itinéraire
  - Liaison des méthodes de recherche d'itinéraire à une API
  - **Reddition intermédiaire (analyse fonctionnelle et journal de bord)**
- 27 octobre 2015:
  - Liaison des méthodes de recherche d'itinéraire à une API
  - Création de l'interface web reposant sur l'API
  - Rédaction de l'analyse organique
- 3 novembre 2015:
  - Création de l'interface web reposant sur l'API
  - Rédaction de l'analyse organique
  - Correction/complétion de l'analyse fonctionnelle
- 10 novembre 2015:
  - Finalisation du code
  - Rédaction de l'analyse organique
  - Correction/complétion de l'analyse fonctionnelle
  - Rédaction d'une conclusion
  - Mise en page de la documentation
- 17 novembre 2015:
  - Finalisation de la documentation
  - Rédaction d'une présentation
  - **Reddition finale**
- 24 novembre 2015:
  - **Présentation**



## Analyse de l'existant

L'idée de programmer une application de recherche d'itinéraire peut paraître inutile: il existe déjà de nombreuses applications qui permettent d'accomplir cette tâche. Cependant, aucune d'entre elles, selon mes recherches, ne traite les données en temps réel.

Les applications existantes se basent sur des données statiques: les tableaux d'horaires, qui ne changent pas en fonction du trafic et du retard des bus, et qui sont mis à jour tous les quelques mois.

L'utilisation des données en temps réel permet d'avoir beaucoup plus de précision sur les heures d'arrivée et de départ des bus, car chaque retard sera répercuté dans les données disponibles. Il est donc possible d'obtenir des itinéraires plus rapides, plus précis et reflétant mieux la réalité en se basant sur ces données.

Cependant, l'utilisation des données en temps réel a aussi ses défauts. Bien que très adaptées pour rechercher un trajet à court terme, elles ne permettent pas de planifier des trajets sur le long terme. En effet, les données disponibles en temps réel ne permettent que de voir les trajets des prochaines heures. Elles sont donc très adaptées pour des recherches d'itinéraires immédiats, par exemple si l'on cherche des routes quelques minutes avant le départ, mais elles ne permettent pas de planifier des itinéraires pour les heures et les jours qui suivent.

Voici quelques services de recherche d'itinéraire pour les TPG:

### Site des TPG

Le [site des TPG](#) permet de rechercher des itinéraires depuis les horaires statiques. L'interface est assez vieille et n'est pas *responsive*<sup>2</sup>. L'affichage des détails d'une route demande deux clics, le second rechargera la page. Notez que l'application mobile redirige vers cette page web pour la recherche d'itinéraire.

### Öff


[Öff](#) est une application Android<sup>3</sup> de recherche de transports dans plusieurs villes du monde, avec entre-autres une fonctionnalité de recherche d'itinéraires. Pour Genève, ce sont aussi les données statiques qui sont utilisées.

Leur interface de vue des itinéraires est plutôt novatrice, mais manque un peu d'esthétisme.

---

2. Le *responsive web design*, ou conception de site web adaptatif, est une technique de mise en page de site web qui va adapter la taille, la forme et la position du contenu en fonction de la taille de l'écran de l'utilisateur.

3. Système d'exploitation pour téléphones mobiles très répandu développé par Google.



Français English




Recherche d'itinéraires Horaire Personnel Arrêt/Gare Affiche horaire


**Votre demande d'itinéraire**

**de:** Cité Lignon **Date:** Ma, 17.11.15  
**à:** Chemin du Bac **Heure:** 13:17 (Départ)


[Modifier cette recherche](#) [Faire une nouvelle recherche](#) [Retour](#)

**Vue d'ensemble** [« plus tôt plus tard »](#)

Détails	Arrêt/gare	Date	Heure	Durée	Chang.	Moyen de transport
<input type="checkbox"/>	Cité Lignon Chemin du Bac	17.11.15	dép. 13:03 arr. 13:07	0:04	0	
<input checked="" type="checkbox"/>	Cité Lignon Chemin du Bac	17.11.15	dép. 13:18 arr. 13:22	0:04	0	
<input type="checkbox"/>	Cité Lignon Chemin du Bac	17.11.15	dép. 13:33 arr. 13:37	0:04	0	

[Détaillez la sélection](#) [Tout détailler](#)  [Imprimer cette page](#)

**Vue détaillée**

Arrêt/gare	Date	Arr.	Dép.	Moyen de transport	Remarques
<a href="#">Cité Lignon</a>	17.11.15		13:18	 <a href="#">Bus 23</a>	Bus Direction: ZIPLO
<a href="#">Chemin du Bac</a>		13:22			

Durée: 0:04, circule Lu - Sa  
Indication: Départ/destination remplacé(e)s par une gare équivalente

FIGURE 1 – Vue des itinéraires sur le site des TPG

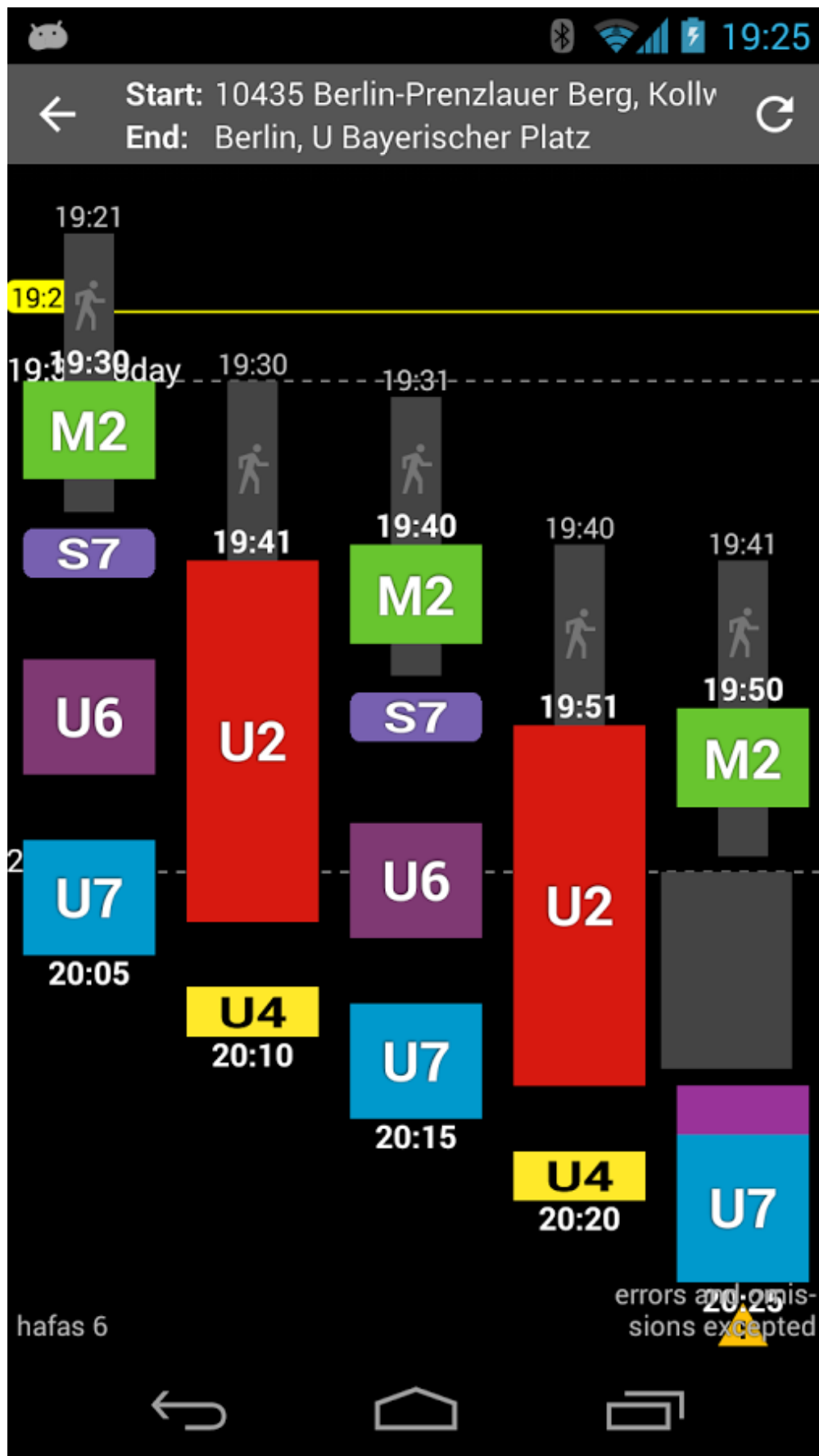


FIGURE 2 – Vue des itinéraires dans l'application Android Öffi

## Itinera

[Itinera](#) est une application iOS<sup>4</sup> de recherche de transports pour Genève, basée sur l'API des transports publics. Cette application a une fonctionnalité de recherche d'itinéraire, mais je ne suis pas en mesure de vérifier si ils sont basés sur les données en temps réel, ne possédant pas d'iDevice<sup>5</sup>. Si c'est le cas, mon projet a toujours l'avantage de fonctionner sur tout appareil possédant un navigateur internet récent.

Vue des itinéraires dans l'application iOS Itinera

## Site des CFF

[Le site des CFF](#) ou leur application mobile permet lui aussi de rechercher des horaires pour les TPG, en plus de la plupart des transports publics de Suisse. Cependant, les données utilisées sont des données statiques, comme avec le site des TPG.

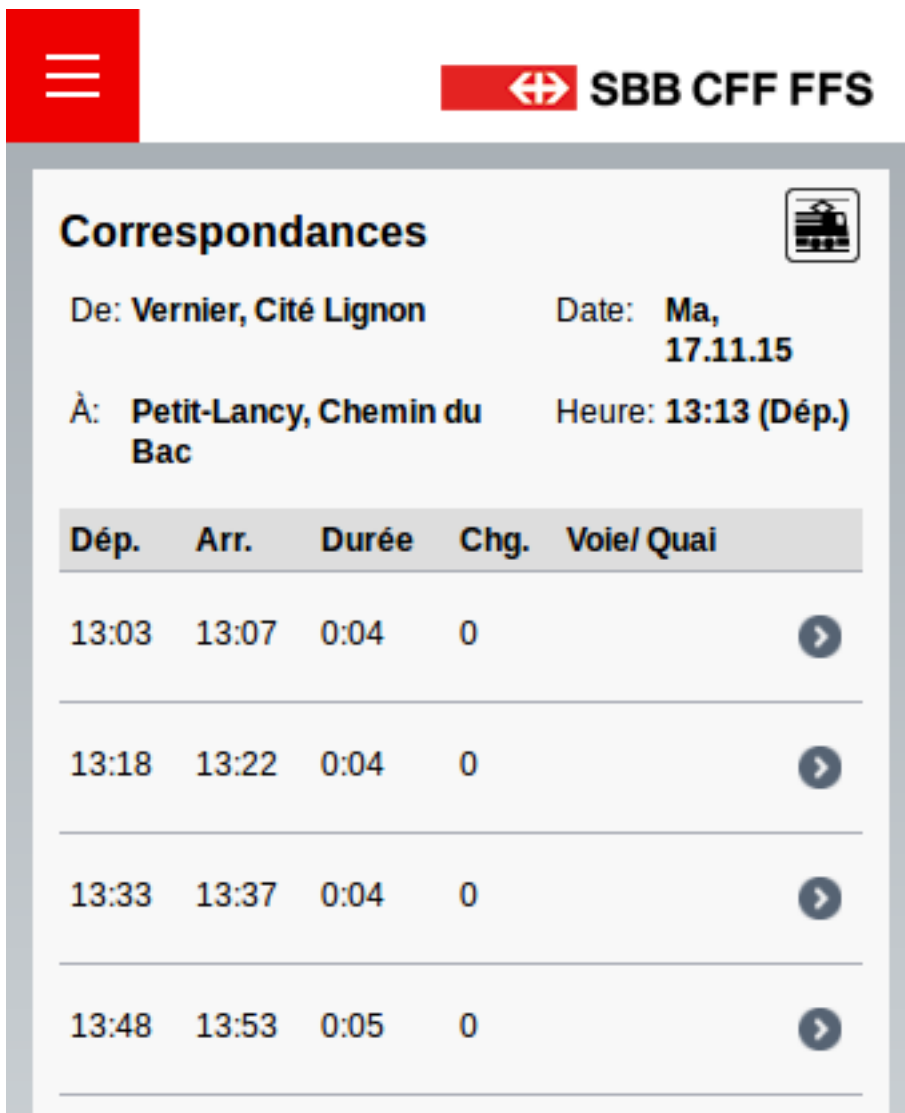
## Conclusion

Je n'ai pas été en mesure de trouver une application permettant de rechercher des horaires en temps réel, en utilisant une interface web, accessible avec n'importe quel navigateur moderne. Le projet TPGraph vient combler ce manque.

---

4. Système d'exploitation pour téléphones mobiles développé par Apple

5. Désigne les appareils d'Apple fonctionnant sur le système iOS, tels que l'iPhone, iPad, iPod Touch.



The image shows a mobile application interface for SBB CFF FFS. At the top, there is a red navigation bar with a white menu icon (three horizontal lines) on the left and the SBB CFF FFS logo on the right. Below the navigation bar, the main content area has a light gray background. The title 'Correspondances' is displayed in bold black text, accompanied by a small train icon. Below the title, the origin and destination are listed: 'De: Vernier, Cité Lignon' and 'À: Petit-Lancy, Chemin du Bac'. The date and time are also shown: 'Date: Ma, 17.11.15' and 'Heure: 13:13 (Dép.)'. A table with five columns (Dép., Arr., Durée, Chg., Voie/ Quai) lists four train correspondences. Each row in the table has a right-pointing arrow icon at the end.

Dép.	Arr.	Durée	Chg.	Voie/ Quai
13:03	13:07	0:04	0	>
13:18	13:22	0:04	0	>
13:33	13:37	0:04	0	>
13:48	13:53	0:05	0	>

FIGURE 3 – Vue depuis le site web (version mobile) des CFF

# Analyse fonctionnelle

## Esquisses de l'interface

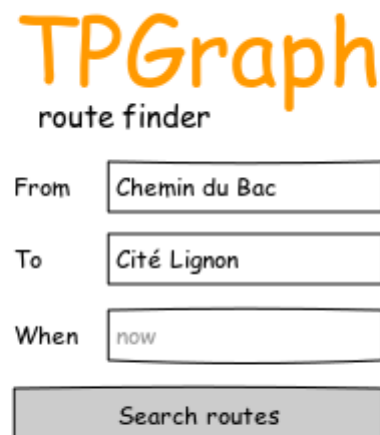
### Page d'accueil

Correspond à la page d'accueil du site. Contiendra un formulaire avec les éléments suivants:

- Arrêt de départ (champ texte avec autosuggestion du nom d'arrêt pendant la frappe)
- Arrêt d'arrivée (champ texte avec autosuggestion du nom d'arrêt pendant la frappe)
- Heure de départ (champ permettant de sélectionner dans les 4 prochaines heures, dont la valeur par défaut est l'heure courante)
- Bouton d'envoi

Cette interface très simple permet à l'utilisateur de sélectionner un arrêt de départ et un arrêt d'arrivée, puis de lancer la recherche.

<http://tpgraph.ch/>



The image shows a sketch of the TPGraph route finder interface. At the top, the text "TPGraph" is written in a large, orange, stylized font, with "route finder" in a smaller, black, sans-serif font below it. Below the text are three input fields. The first field is labeled "From" and contains the text "Chemin du Bac". The second field is labeled "To" and contains the text "Cité Lignon". The third field is labeled "When" and contains the text "now". Below these fields is a grey button with the text "Search routes".

FIGURE 4 – Esquisse de la page home

### Page d'affichage des résultats de recherche d'itinéraire

Correspond aux résultats d'une recherche d'itinéraire.

<http://tpgraph.ch/getroute/?from=cbac&to=clig>

TPGraph

From Chemin du Bac

To Cité Lignon

When 22:53

Get previous routes

🕒 22:56 → 23:29 ⏰ 33 min ↔ 1

🕒 23:17 → 23:34 ⏰ 17 min ↔ 0

Get next routes

FIGURE 5 – Mockup de la page find\_paths

<http://tpgraph.ch/getroute/?from=cbac&to=clig>

TPGraph

From Chemin du Bac

To Cité Lignon

Get previous routes

🕒 22:56 → 23:29 ⏰ 33 min ↔ 1

7	Cité Lignon	22:56	17 min
	Bel-Air	23:13	
🚶	Bel-Air	23:13	2 min
	Bel-Air	23:15	
18	Bel-Air	23:17	12 min
	Bouchet	23:29	

🕒 23:17 → 23:34 ⏰ 17 min ↔ 0

Get next routes

FIGURE 6 – Mockup de la page find\_paths avec les détails d'un chemin

## Base de données orientée graphe

Les systèmes de gestion de bases de données graphe font partie de la famille *NoSQL*, qui signifie *Not only SQL* (pas seulement SQL). Cette famille regroupe différents systèmes de bases de données ne classant pas les données selon un modèle pré-établi et difficilement modifiable en production, au contraire des bases de données *relationnelles*.

Une base de données orientée graphe structure ses données dans un graphe, au sens mathématique du terme. Un graphe est composé de noeuds (ou *vertices* en anglais) reliés entre eux par des arcs (ou *edges* en anglais). Dans le monde de l'informatique, on parle souvent de *relation* au lieu d'utiliser le mot *arc*.

Les relations peuvent être dirigées ou non, c'est à dire qu'elles peuvent pointer d'un noeud vers un autre avec ou sans une indication de la direction de la relation.

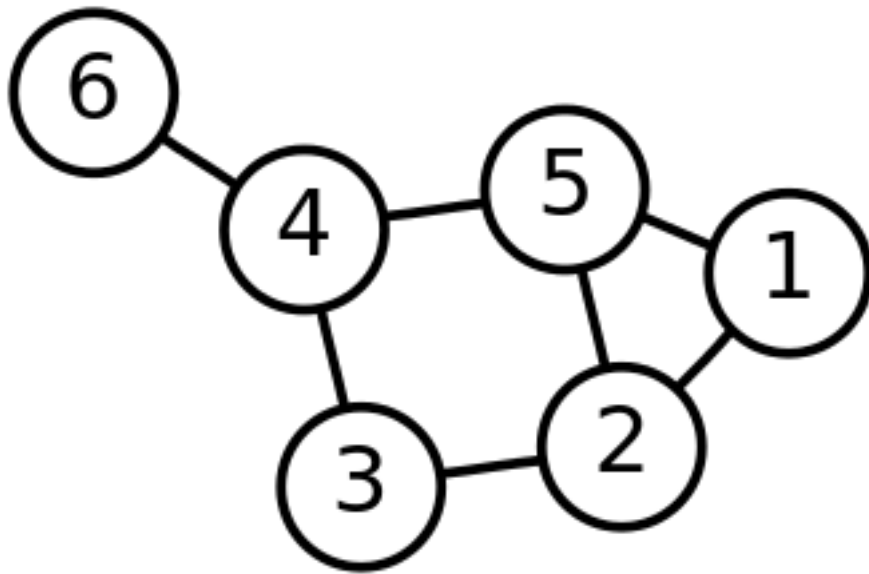


FIGURE 7 – Exemple de graphe avec des relations qui ne sont pas dirigées

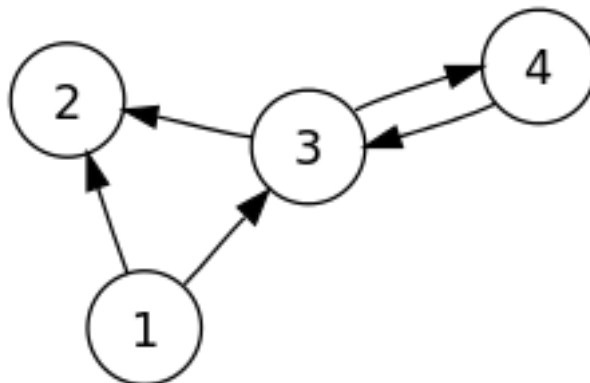


FIGURE 8 – Exemple de graphe avec des relations dirigées



Les données sont stockées dans des couples clé-valeur appelés propriétés. Ces propriétés peuvent être assignées aussi bien à des noeuds qu'à des relations.

La grande force de ces bases de données est qu'elles permettent de faire des traversées complexes à travers des longues suites de relations en très peu de temps. Elles permettent d'utiliser des langages spécialisés permettant d'exprimer facilement ces requêtes qui seraient très difficiles à effectuer en SQL avec une base de données relationnelle traditionnelle.

## Neo4j

Neo4j est un système de gestion de base de données (SGBD) NoSQL orienté graphe, implémenté en Java. C'est la base de données orientée graphe la plus populaire au moment de l'écriture de ces lignes.

Neo4j est développée par Neo Technology, Inc., basée à San Francisco aux États-Unis d'Amérique et à Malmö en Suède.

Il existe une version open-source et une version commerciale. C'est la version open-source qui sera utilisée dans ce travail. La version commerciale apporte des fonctionnalités de clustering qui dépassent les besoins de ce projet.

## Framework web Django

Le framework Django est un framework web écrit en python. Il sous licence open-source, et est développé par la Django Software Foundation, qui est une organisation à but non lucratif.

Le framework est structuré selon le principe *Model-Template-View* (MTV), qui est fondamentalement similaire au *Model-View-Controller* (MVC), sauf que les *controllers* du MVC sont nommés *views* en MTV, et que les *views* du MVC sont nommés *templates*. Les développeurs de Django partent en effet du principe que le mot *view* représente plutôt une vue spécifique des données, qui peut être ensuite rendue en HTML, PDF, JSON, XML, etc. Il s'agit surtout d'une différence de vocabulaire, car un *controller* en MVC est similaire à une *view* en MTV, et une *view* en MVC est similaire à un *template* en MTV.

Django fournit un moteur d'*Object-Relational-Mapping* (ORM), qui permet une abstraction de la base de données en objets dans le code. Ainsi, on crée un modèle avec des classes contenant différents champs, et l'ORM va se charger de fabriquer une base de données correspondant à ce modèle objet, et permet ensuite d'utiliser les objets pour écrire et lire dans la base, sans ne jamais taper une ligne de SQL.

Cependant, pour quelques requêtes complexes dont les performances sont critiques, il est parfois plus avisé d'écrire ses requêtes à la main. Ces requêtes peuvent être encapsulées dans les méthodes des objets du modèle.

Étant donné que l'application TPGraph se basera sur une base de données orientée graphe, elle n'utilisera pas l'ORM, qui pourra toujours être utilisé dans des versions futures, par exemple pour la mémorisation de comptes utilisateurs, domaine dans lequel les bases de données relationnelle n'ont plus à prouver leur efficacité.

## Docker

Docker est un programme qui permet d'automatiser le déploiement d'applications/services dans des conteneurs logiciels autonomes. Ces conteneurs permettent d'isoler les ressources de chaque application/service, sans avoir à créer de machine virtuelle pour chacune. Docker utilise les fonctionnalités du noyau GNU/Linux pour créer des conteneurs isolés. Ainsi, les ressources de l'hôte sont utilisées de manière optimale, contrairement à l'utilisation de machines virtuelles qui sont très gourmandes.

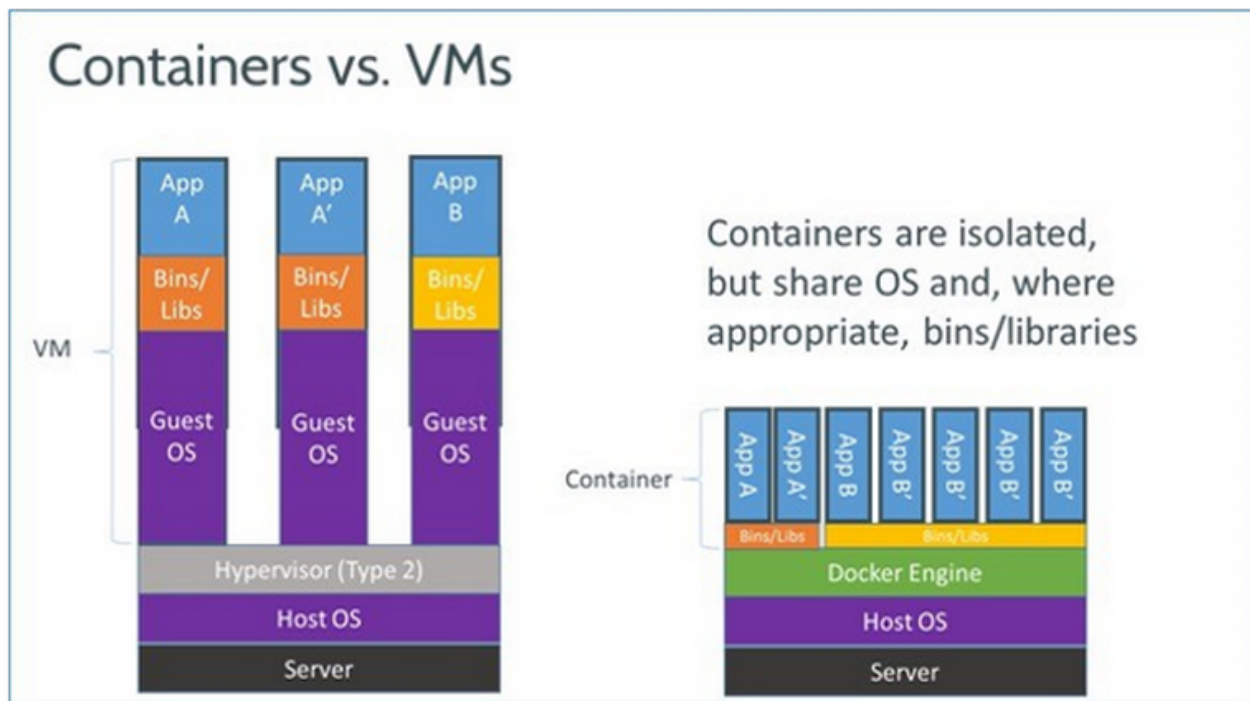


FIGURE 9 – Comparaison entre les machines virtuelles et les containers.

L'outil Docker Compose sera utilisé pour définir et interconnecter les différentes applications/services dont TPGraph a besoin pour fonctionner. Il s'agit en quelque sorte d'une "recette" qui permet de mettre en place toute l'architecture de l'application et ses dépendances très simplement et rapidement sur n'importe quelle machine. Cela permet de simplifier grandement le développement depuis plusieurs machines, car il est très simple de reconstruire toute l'architecture de l'application.

Cet outil peut simplifier grandement la mise en production de l'application sur un serveur. Il suffit en effet d'installer docker et docker-compose sur le serveur, d'exécuter la "recette" pour que l'application soit installée et lancée.

L'outil Docker Machine sera lui utilisé pour garder docker dans une machine virtuelle commandable à distance, afin de ne pas interférer avec l'hôte. Si l'application devra un jour être déployée sur un serveur distant, Docker Machine permettra d'automatiser cette tâche.

# Analyse organique

## Structure des containers Docker

Le fichier `docker_compose.yml` contient la *recette de cuisine* qui permet de monter l'application dans les différents containers appropriés. Les connexions entre les conteneurs, ainsi que les ports ouverts vers l'extérieur sont définis dans ce fichier.

- **web** : Contient l'application *TPGraph*, ainsi que le serveur d'application *Gunicorn* qui gèrera l'exécution des différentes requêtes dynamiques passées à l'application. Se base sur le dossier `web/` comme source de données initiales.
- **nginx** : serveur web par lequel transiteront toutes les requêtes entrantes. Les requêtes de ressources dynamiques seront transmises à *Gunicorn*, tandis que les requêtes de ressources statiques seront transmises directement. Se base sur le dossier `nginx/` comme source de données initiales.
- **redis** : Base de données clé-valeur en mémoire très performante. Elle sera utilisée comme *backend* pour la librairie python *Celery* qui permet de gérer des tâches de fond sur le serveur, en l'occurrence, le rafraîchissement ponctuel de la base de données.
- **neo4j** : Base de données orientée graphe dans laquelle les données des TPG seront stockées.
- **postgres**: Base de données relationnelle. Elle est là pour gérer les comptes utilisateurs de Django, même si cette fonctionnalité n'est pas utilisé dans la version actuelle de l'application.
- **data**: contient les données de **postgres** dans un conteneur séparé, ce qui permet de réinitialiser le container de la base de données sans perdre les données.
- **celeryworker**: *Worker* qui va exécuter les tâches de fond de l'application. Se base sur le dossier `web/` comme source de données initiales.
- **flower**: interface de gestion des tâches de fond et des *workers*. Se base sur le dossier `flower/` comme source de données initiales.

## Structure du graphe de l'application

Afin de pouvoir calculer les itinéraires, les données ont été structurées judicieusement.

J'ai décidé d'utiliser 5 types de nœuds décrits ci-dessous. Dans la liste des propriétés de chaque nœud, celles marquées en gras ont une contrainte d'unicité, c'est à dire que deux nœuds du même type ne peuvent pas avoir la même valeur pour cette propriété. Les propriétés marquées comme uniques sont automatiquement indexées. Cela permet d'accélérer grandement la recherche d'un nœud par cette propriété.

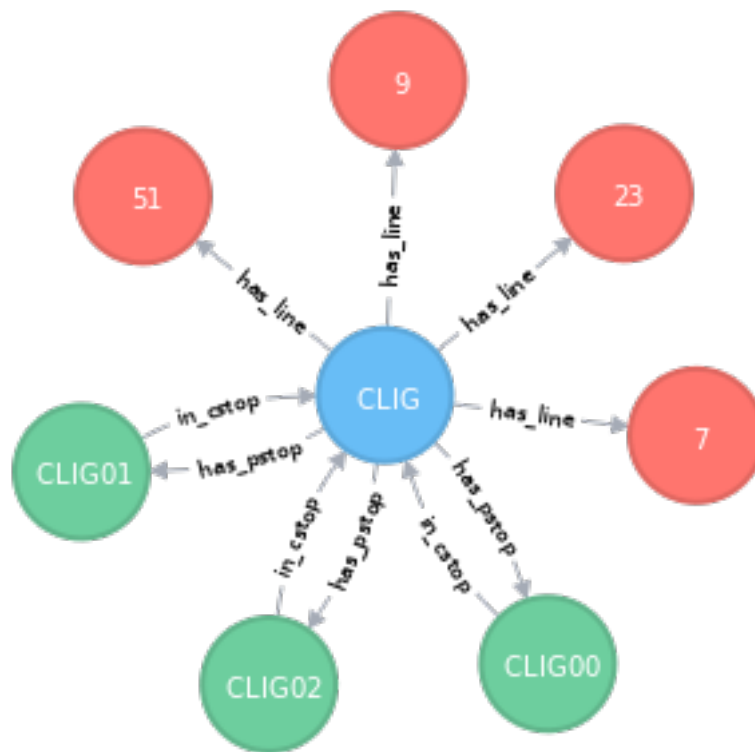


FIGURE 10 – Exemple de noeud `commercial_stop` (en bleu), lié à des `physical_stop` (en vert) et des `line` (en rouge)

### `commercial_stop`

Représente un arrêt commercial, qui est un regroupement de plusieurs arrêts physiques portant le même nom.

Par exemple, l'arrêt *Cornavin* regroupe différents arrêts physiques qui se situent à des endroits différents.

Propriétés:

- **code** : Code de 4 caractères de l'arrêt. Exemple: *CLIG*
- **name** : Nom complet de l'arrêt. Exemple: *Cité Lignon*

Relations sortantes:

- **has\_pstop** : Dirige vers les `physical_stops` que regroupe cet arrêt.
- **has\_line** : Dirige vers les `lines` faisant escale à cet arrêt.

Relations entrantes:

- **in\_cstop** : Dirigé depuis les mêmes arrêts que **has\_pstop**. Ces relations ont été créées afin de simplifier la recherche d'itinéraire qui se fait sur des relations dirigées dans un sens uniquement. Il fallait donc avoir un chemin de retour pour passer d'un `physical_stop` à un autre se trouvant dans le même `commercial_stop`.

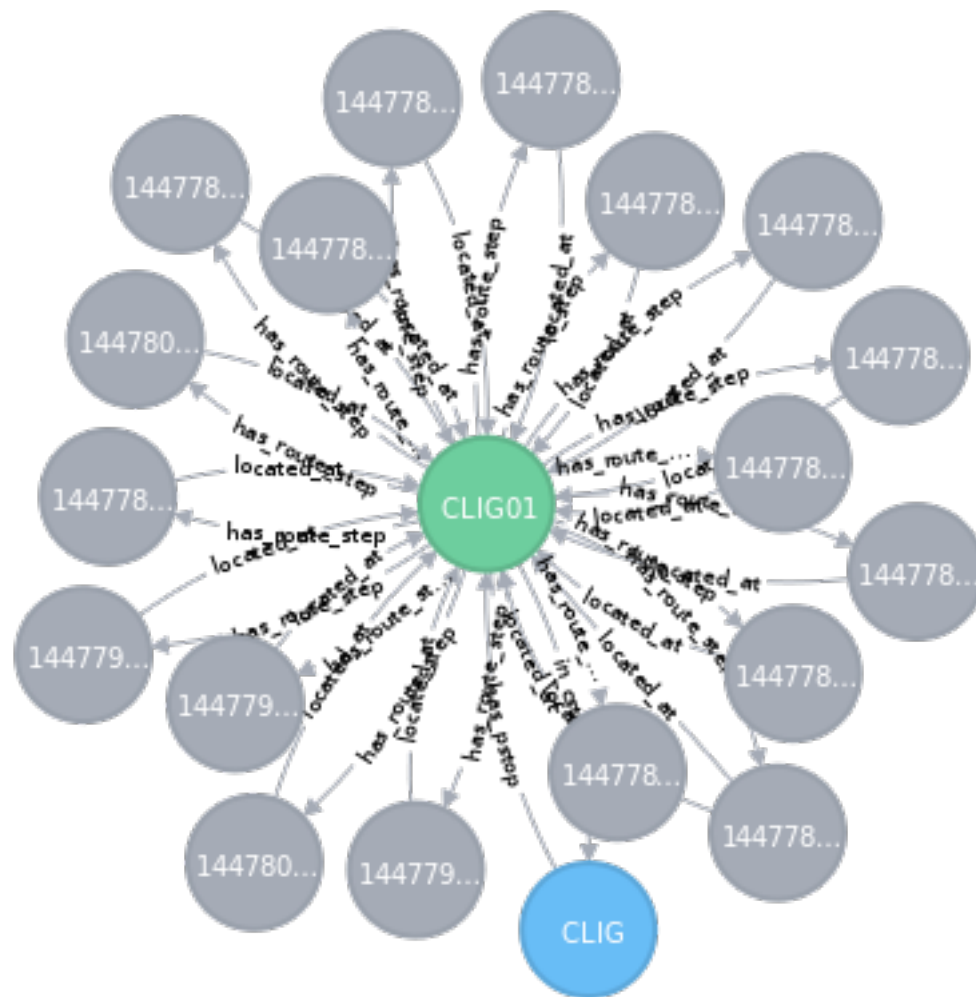
**physical\_stop**

FIGURE 11 – Exemple de noeud `physical_stop` (en vert), lié à des `route_step` (en gris) et un `commercial_stop` (en bleu)

Représente un arrêt physique, avec ses coordonnées géographiques. Ces dernières ne sont actuellement pas utilisées mais pourraient servir à calculer des liaisons entre les arrêts proches géographiquement, afin d'implémenter les itinéraires qui comportent des trajets à pied entre deux arrêts (qui ne font pas partie du même `commercial_stop`).

Propriétés:

- **code**: Code de l'arrêt. Souvent formé à partir du code de l'arrêt physique + un numéro. Exemple: *CLIG01*
- **name**: Nom complet de l'arrêt. Souvent similaire à celui de l'arrêt commercial lié. Exemple: *Cité Lignon*
- **lon**: Longitude de la position géographique de l'arrêt.
- **lat**: Latitude de la position géographique de l'arrêt.

Relations sortantes:

- `has_route_step` : Dirige vers les `route_step` passant par cet arrêt.
- `in_cstop` : Dirige vers le `commercial_stop` correspondant.

Relations entrantes:

- `located_at` : Dirigé depuis les `route_step` passant par cet arrêt. Doublon de `has_route_step` mais dirigé dans le sens inverse.
- `has_pstop` : Dirigé depuis le `commercial_stop` correspondant. Doublon de `in_cstop` mais dirigé dans le sens inverse.

**line**

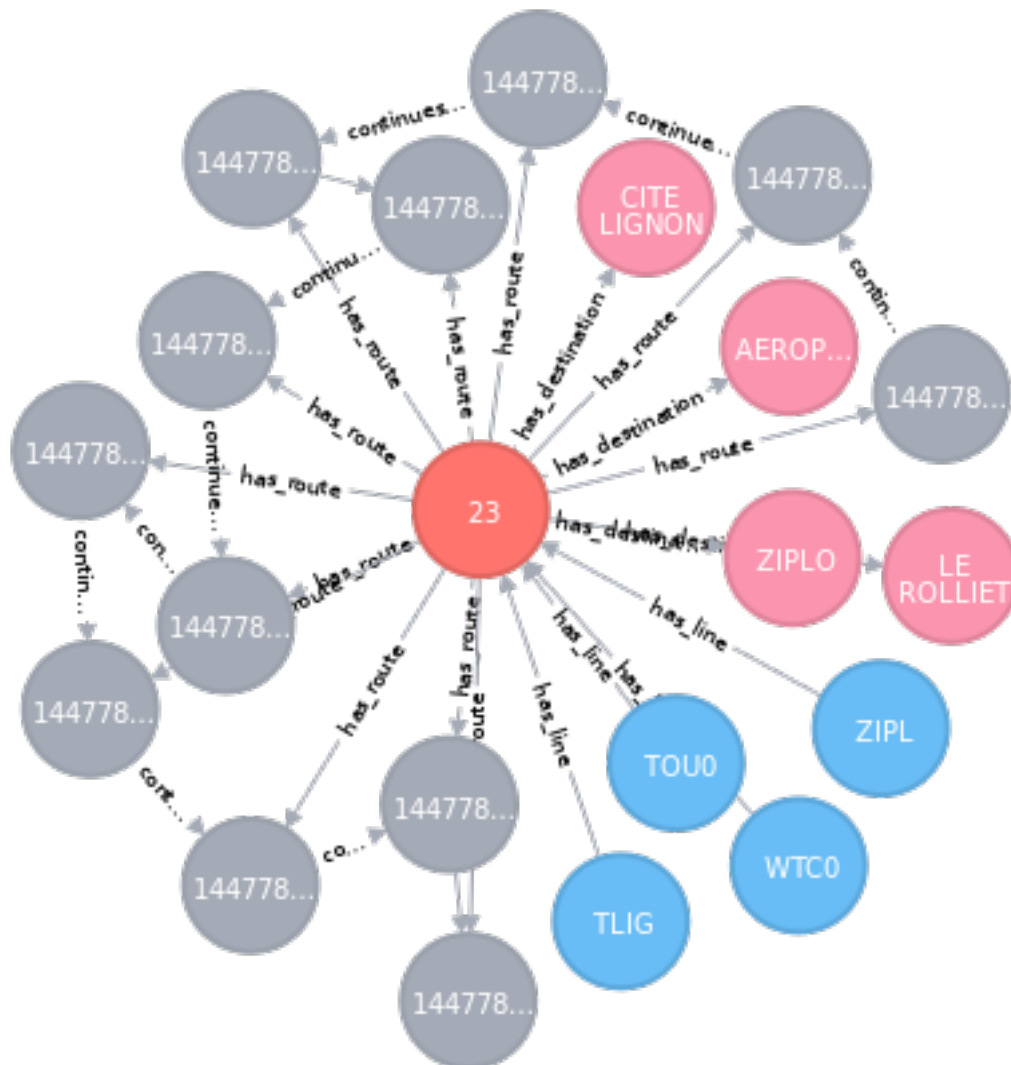


FIGURE 12 – Exemple de noeud `line` (en rouge), lié à des `route_step` (en gris), des `destination` (en rose) et des `commercial_stop` (en bleu)

Représente une ligne de bus, indépendamment de la direction.

Propriétés:

- **code** : Code (nom) de la ligne. Exemple: *23*

Relations sortantes:

- **has\_destination** : Dirige vers les **destination** de cette ligne. (inutilisé pour le moment)
- **has\_route** : Dirige vers une **route\_step** desservie par cette ligne. Utilisé pour savoir à quelle ligne appartient une **route\_step**.

Relations entrantes:

- **has\_line** : Dirigé depuis les **commercial\_stop** correspondants.

### **route\_step**

Représente une étape d'une route. Si une ligne s'arrête à 15 arrêts sur une route donnée, il y aura donc 15 steps. Une route est tout simplement une suite de noeuds **route\_step** liés par de relations **continues\_with**.

Propriétés:

- **departure\_code**: Code de départ unique. Exemple: *182365*

Relations sortantes:

- **continues\_with** : Dirige vers la **route\_step** suivante sur la route.
- **located\_at** : Dirige vers le **physical\_stop** desservi par cette étape.

Relations entrantes:

- **has\_route\_step**: Dirigé depuis le **physical\_stop** desservi par cette étape. Doublon de **located\_at** mais dirigé dans le sens inverse.
- **has\_route**: Dirigé depuis la **line** effectuant cet arrêt.

### **destination**

Représente une destination de ligne. Étrangement, dans les données de l'API des TPG, une destination est une entité différente qu'un **commercial\_stop** ou un **physical\_stop**. Son but est d'indiquer le sens d'une ligne.

Ces noeuds ont été intégrés au graphe mais ne sont pour l'instant pas utilisés par l'application.

Propriétés:

- **code**: Nom de la destination

Relations entrantes:

- **has\_destination** : Dirigé depuis une ligne desservant cette destination.

## **Requête d'itinéraires dans le graphe**

Une fois que le graphe contient les données récupérées *via* l'API des TPG, il faut pouvoir récupérer les chemins les plus courts, en utilisant le langage *Cypher*.

Ici, la structure de données détaillée plus haut prendra tout son sens:

Il faut trouver une requête *Cypher* qui permettra de trouver les chemins les plus courts (temporellement) entre deux `commercial_stop`.

Le chemin entre deux arrêts pour une date de départ T donnée peut être défini selon les contraintes suivantes:

Contraintes topologiques:

1. Le chemin commence par un noeud de type `commercial_stop`.
2. Le chemin se termine par un noeud `commercial_stop`.
3. Un nombre indéfini de relations orientées, de type `located_at`, `continues_with`, `in_cstop`, `has_pstop` ou `has_route_step` séparent le noeud de départ et celui d'arrivée.

Contraintes chronologiques:

4. Le premier noeud `route_step` dans le chemin doit avoir une date de départ supérieure à la date de départ T.
5. La date de départ (déduite du premier `route_step`) est avant la date d'arrivée (déduite du dernier `route_step`)
6. La date de départ du noeud `route_step` du début de chaque connexion (changement de bus) doit être inférieure à la date de départ du dernier noeud `route_step` rencontré.
7. Les résultats les plus pertinents sont ceux dont la date du dernier `route_step` est la plus basse, puis ceux dont la durée est la plus .

Les contraintes 1, 2 et 3 peuvent être implémentées ainsi :

```
MATCH AllShortestPaths(
  (start_cstop:commercial_stop { code: {start_cstop_code} } )
  -[path_rels
    :located_at |
    :continues_with |
    :in_cstop |
    :has_pstop |
    :has_route_step
  *]->
  (end_cstop:commercial_stop { code: {end_cstop_code} } )
)
```

Pour pouvoir tester les contraintes suivantes, on récupère chaque connexion (changement de bus) dans le chemin:

```
WITH [
  // foreach relation between physical_stop and route_step
  r IN path_rels
  WHERE ()-[r:located_at|:has_route_step]->() |
```



```

// group the route step and the physical stop in a dict
CASE
  WHEN ()-[r:located_at]->() THEN
  {
    route_step: STARTNODE(r),
    physical_stop: ENDNODE(r)
  }
  WHEN ()-[r:has_route_step]->() THEN
  {
    route_step: ENDNODE(r),
    physical_stop: STARTNODE(r)
  }
END
]
AS connections

```

Puis on en déduit la date de départ et d'arrivée, tout en gardant les connexions de côté pour la suite:

```

WITH HEAD(connections).route_step.timestamp AS departure_time,
      LAST(connections).route_step.timestamp AS arrival_time,
      connections

```

On peut maintenant appliquer les condition 4 et 5:

```

WHERE departure_time < {after_timestamp}
AND departure_time < arrival_time

```

La condition 6 est un peu moins élégante à mettre en œuvre, mais les performances restent bonnes. On va parcourir chaque connexion, en se souvenant du timestamp de la précédente. La première valeur est arbitrairement fixée à 0. Si on se retrouve avec une date inférieure à la précédente, on retourne -1. Si la date précédente est -1, on n'essaye plus de la comparer avec la courante et on retourne -1. Une fois l'itération terminée, la dernière valeur est retournée. Si elle est supérieure ou égale à 0, on peut conclure que l'ordre des connections est bien chronologique.

```

AND REDUCE(
  last = 0,
  c IN connections |
  CASE
    WHEN last >= 0 AND c.route_step.timestamp-last >= 0 THEN
      c.route_step.timestamp
    ELSE
      -1
  END
) >= 0

```

On peut ensuite retourner les valeurs:

```
RETURN connections,  
       arrival_time-departure_time AS duration,  
       departure_time, arrival_time  
  
ORDER BY arrival_time,  
         duration  
LIMIT 10
```

## Application Django

### Modèle

Il existe un module django nommé *Neo4Django* qui permet de faire des requêtes dans une base de données *Neo4j* en utilisant la même syntaxe que l'ORM Django. Cela permet de récupérer de manière très simple des données provenant de Neo4j. Cette librairie n'étant malheureusement pas disponible pour Python 3, le modèle a donc été implémenté *à la main*, en utilisant une librairie plus bas-niveau pour communiquer avec Neo4j: *py2neo*.

Un mapping a été effectué entre les différents types de noeuds utilisés dans le graphe et des classes, qui permettent chacune de récupérer un ou plusieurs objets.

- `CommercialStop`
- `PhysicalStop`
- `RouteStep`
- `Line`

Des classes `Route` et `RouteConnection` ont aussi été créées pour pouvoir lancer et représenter les requêtes d'itinéraire.

### Views

La vue est la partie de l'application qui va récupérer des données provenant du *model*, et les lier à un *template*.

L'application comprend 3 *views*:

**home** Affiche la page d'accueil avec les différentes métriques.

Cette vue est rendue avec le template `home.html`.

**find\_paths** Affiche les résultats d'une recherche d'itinéraire.

Cette vue est rendue avec le template `paths.html`.

## TPGraph

From

To

Search routes.

811	1706
Commercial Stops	Physical Stops
38975	71
Route Steps	Lines

FIGURE 13 – Aperçu du rendu de la vue home

# TPGraph

**From:**

Cité Lignon

**To:**

Bout-du-Monde

**When:**

11/17/2015 9:51 p.m.

🕒 19:05 ➔ 19:39			⌚ 33:26			🔄 1		
Line			Stop			Time		
23			Cité Lignon			19:05		
			Bossons			19:13		
21			Bossons			19:13		
			Bout-du-Monde			19:39		

FIGURE 14 – Aperçu du rendu de la vue `find_paths`

**populate\_graph** Lance la tâche de population du graphe manuellement.

Cette vue n'a pas de rendu, elle redirige sur **home**.

## Routage

Les URLs<sup>6</sup> de l'application sont les suivants:

- `http://tpgraph.ch/` : dirige vers la vue **home**
- `http://tpgraph.ch/find_paths/` : dirige vers la vue **find\_paths**
- `http://tpgraph.ch/populate_graph/` : dirige vers la vue **populate\_graph**

Note: l'url `http://tpgraph.ch` est indicative et peut être substituée par une autre pendant le développement. Cependant, j'ai fait l'acquisition de ce nom de domaine et j'ai la possibilité de l'utiliser si l'application est assez stable pour être déployée en production à la fin du projet.

## Tâches *Celery*

*Celery* est une librairie python qui permet de lancer des tâches de fond, exécutées par un processus spécialisé appelé *worker*.

Quand une tâche est lancée depuis une vue, par exemple, un message demandant au *worker* de commencer la tâche est envoyé. Ces messages transitent via *Redis*, qui est une base de données qui reste uniquement en mémoire, ce qui assure des performances très hautes. Le *worker* commencera la tâche quand sa file d'attente se videra.

Il est aussi possible de planifier les tâches afin qu'elles s'exécutent périodiquement.

L'application TPGraph contient 2 tâches périodiques, définies dans `tasks.py`:

- **populate\_graph\_task** : lance la population du graphe avec les données des TPG. Les noeuds et les relations existants seront mis à jour ou resteront inchangés s'il n'y a rien à modifier, tandis que les nouveaux noeuds et les nouvelles relations seront créées. Cette tâche est lancée toutes les 5 minutes.
- **clean\_graph\_task** : supprime les routes du graphe qui sont dans le passé, c'est à dire celles dont la date est avant la date courante. Cette tâche est aussi lancée toutes les 5 minutes, mais avec une priorité supérieure à **populate\_graph\_task**. Cette priorité a été décidée car cette tâche est beaucoup plus rapide à exécuter (quelques secondes).

---

6. *Uniform Resource Locator*: adresse permettant d'accéder à une ressource

# Guide de maintenance

## Installation de l'application en local

Cette procédure d'installation a été testée sous Ubuntu Linux, 14.04, 15.04 et 15.10.

Il est requis d'avoir installé le système en 64 bits et avoir un kernel Linux supérieur ou égal à la version 3.10. Note: Les utilisateurs de MacOSX et de Windows peuvent installer [Docker Toolbox](#), qui contient les trois dépendances ci-dessous, mais le support de ces plates-formes n'a pas été testé.

1. [Installer Docker Engine](#)
2. [Installer Docker Compose](#)
3. [Installer Docker Machine](#)
4. Exécuter le script `./tpgraph-docker/create_machine.sh`

Le script fabrique une machine virtuelle nommée *dev* avec l'outil Docker Machine, puis Docker Compose est exécuté sur cette machine pour initialiser et lancer les différents conteneurs requis par l'application. L'exécution du script peut être longue si c'est la première fois, car docker doit télécharger des images sur DockerHub <sup>7</sup>.

La dernière ligne affichée par le script `create_machine.sh` permet de connaître l'adresse IP de la machine virtuelle et de se connecter au site à l'aide d'un navigateur web. Par exemple:

```
Machine started with ip 192.168.99.100
```

Il est tout à fait possible de ne pas utiliser Docker Machine et de lancer les containers depuis Docker Compose en local, sur la machine hôte, sans passer par une VM <sup>8</sup>. Docker Machine offre cependant plus de flexibilité, car il permet de ne rien changer sur la machine hôte et de réinitialiser complètement la machine en cas de besoin.

De plus, Docker Machine fournit des drivers qui permettent très simplement de déployer docker sur des serveurs distants, par exemple pour la mise en production. Je n'ai malheureusement pas encore pu tester cette fonctionnalité mais il est prévu de l'utiliser pour un éventuel déploiement futur de l'application sur le web.

## Mise à jour de l'application après modification des sources

Si le script est relancé une seconde fois mais que la machine virtuelle existe déjà, elle ne sera pas recrée, et Docker Compose ne mettra à jour que ce qui a changé. Cette opération dure moins de 5 secondes si les modifications n'impliquent pas le téléchargement d'un nouveau

---

7. Dépôt officiel de containers Docker, contenant des images pour tous la plupart des services en vogue.

8. Machine virtuelle

container. Le script peut donc être relancé à chaque fois que le code est modifié pour mettre à jour l'application.

Quand le script est relancé, la base de données reste inchangée et les données sont toujours là. Pour réinitialiser tous les conteneurs, il suffit de lancer les commandes suivantes:

```
# Évalue l'environnement de la machine 'dev', ce qui redirigera les commandes
# docker-compose sur la VM créée par docker-machine au lieu de les lancer
# en local
eval $(docker-machine env dev)

# Extinction de tous les conteneurs sur 'dev'
docker-compose kill

# Suppression de tous les conteneurs sur 'dev'
# Note: les fichiers d'initialisation des conteneurs sont mis en cache et
# ils ne seront pas téléchargés à nouveau
docker-compose rm

# Régénération et lancement des conteneurs
./create_machine.sh
```

Si vous souhaitez recréer la machine virtuelle au complet, ce qui implique que tous les fichiers seront téléchargés à nouveau depuis DockerHub, lancez les commandes suivantes:

```
docker-machine kill dev
docker-machine rm dev
./create_machine.sh
```

## Utilisation de l'application

### Accès à l'interface web

Rendez-vous sur l'adresse de l'application, par exemple sur <http://192.168.99.100/> (attention, cette adresse peut varier, il faut se référer à l'adresse retournée par le script `create_machine.sh`).

Une page devrait apparaître.

### Description de la page d'accueil

Cette dernière contient un formulaire de recherche d'itinéraire très simple composé de:

- Un champ pour entrer l'arrêt de départ, libellée *From*. Ce champ bénéficie de complétion automatique du nom d'arrêt.

- Une champ pour entrer l'arrêt d'arrivée, libellée *To*. Ce champ bénéficie de complétion automatique du nom d'arrêt.
- Un bouton permettant de lancer la recherche.

En dessous du formulaire de recherche d'itinéraire, quatre valeurs sont affichées:

- Nombre d'arrêts commerciaux (*Commercial Stops*)
- Nombre d'arrêts physiques (*Physical Stops*)
- Nombre d'étapes de routes (*Route Steps*)
- Nombre de lignes (*Lines*)

Si vous venez d'allumer l'application, il est possible que tous ces indicateurs soient à zéro. C'est normal: les données n'ont pas encore été chargées. Tant que les données ne sont pas chargées, il n'est pas possible d'effectuer une recherche d'itinéraire.

## Chargement des données

Il y a deux moyens de charger les données:

- Méthode automatique: l'application lance une tâche de fond de chargement des données toutes les 5 minutes. Il suffit d'attendre.
- Méthode manuelle: se rendre sur la page `/populate_graph/` du site, ce qui lancera la tâche de fond de chargement des données manuellement. Cette page existe à des fins d'aide au développement et pourrait être supprimée dans des versions ultérieures de l'application.

Il n'y a pas à se préoccuper du nombre de tâches en cours: seule une tâche est autorisée à être exécutée à la fois. Si deux tâches sont lancées en même temps, elles s'exécuteront l'une après l'autre.

On peut constater que les données sont chargées quand le nombre d'étapes de routes (*Route Steps*) atteint entre les 100'000 et 200'000 entrées et que ce chiffre n'augmente plus rapidement. Ce chiffre peut néanmoins varier selon les jours et les heures. Il y a en effet moins de bus le soir et les weekends.

## Recherche de routes

Une fois les données chargées, vous pouvez taper le nom de deux arrêts de bus dans les champs prévus à cet effet, puis cliquer sur le bouton intitulé *Search routes* (rechercher des routes).

Une nouvelle page devrait s'ouvrir, avec 10 résultats de recherche (au maximum). Il n'est malheureusement pas possible d'afficher plus de résultats dans l'état actuel de l'application.

## Interfaces secondaires

L'application repose sur deux services essentiels à son fonctionnement: *Celery* et *Neo4j*. Ces deux services disposent d'interfaces permettant de les administrer et/ou d'analyser leur fonctionnement.



En production, ces pages devraient être désactivées ou protégées par un système d'authentification. Le site étant à l'état de prototype, ces pages sont accessibles en clair.

### Monitoring des tâches de fond : *Flower*

Il est possible de visionner l'état des *workers* Celery depuis une interface web nommée *Flower*. Elle est accessible à la même adresse que le site TPGraph, mais sur le port 5555. Par exemple: <http://192.168.99.100:5555/>.

Depuis cette interface, on peut voir quelle est la/les tâches en cours, celles en attente, le *message broker* utilisé, etc. C'est très utile pour le développement et le débogage des tâches.

### Interfaces d'administration de Neo4J

Neo4J met à disposition une interface web très pratique pour tester des requêtes sur le graph, modifier les données, ou encore vérifier l'état du serveur.

Elle est accessible à la même adresse que le site TPGraph, mais sur le port 7474. Par exemple: <http://192.168.99.100:7474/>.

## Protocole de test

Étant donné que le temps était limité en fin de projet, j'ai testé le bon fonctionnement de l'application en comparant les itinéraires retournés avec les horaires en temps réel des TPG. Les dates retournées sont bien les bonnes.

J'ai aussi testé que l'accès à chaque vue était fonctionnel et que chaque tâche s'exécutait correctement.

Il va de soi qu'un protocole plus élaboré devrait être élaboré pour les versions futures de l'application.

# Conclusion

## Retour sur la planification

Ce projet était très intéressant, mais aussi très ambitieux. La planification initiale, décrite dans le cahier des charges, a été plus ou moins respectée durant les premières semaines, mais des problèmes techniques et la recherche d'information sur des technologies que je ne maîtrisais pas (Docker et Neo4j) ont ensuite beaucoup ralenti le développement.

Le fait d'avoir commencé à rédiger la documentation dès le début du travail a ensuite permis de consacrer beaucoup de temps au développement. Étant donné que les technologies utilisées étaient inconnues, les différentes étapes du projet ont demandé plus de temps que prévu, mais cela a permis d'acquérir beaucoup de nouvelles compétences qui permettront de travailler plus efficacement sur des projets futurs.

## Retour sur les technologies

### Docker

Docker est une technologie nouvelle et prometteuse, mais la maîtrise de cet outil n'a pas été une chose aisée, car il y a encore relativement peu de documentation à son sujet, et certaines informations ne sont plus valides car elles concernent des versions antérieures.

Cependant, après beaucoup d'heures passées à peaufiner sa configuration, j'ai appris à l'utiliser correctement et je suis en mesure d'affirmer qu'il s'agit d'un outil extrêmement puissant, qui s'est montré très pratique pour le développement depuis plusieurs postes. Une éventuelle mise en production sera d'ailleurs grandement simplifiée grâce à lui.

Je n'ai absolument pas de regrets quand à l'intégration de cette technologie dans mon projet, malgré tout le temps que cela a pris pour la configurer. Grâce à Docker, il est trivial de remonter toute la pile de technologies nécessaire à l'exécution du projet, pour n'importe quel développeur, sur n'importe quelle machine.

Cette technologie émergente fait beaucoup de bruit aujourd'hui et il y a fort à parier qu'elle sera un standard d'ici moins d'une dizaine d'années, comme les laissent présager les récents partenariats entre Docker et de grosses entreprises comme Microsoft, Red Hat et Amazon.

### Django

Django a été un framework tout à fait adéquat pour ce projet, bien que l'ORM n'ait pas été utilisé. L'utilisation de ce framework, que je connaissais déjà avant le début du travail, a permis de développer très rapidement toute la partie *web* de l'application, et de se concentrer sur les tâches plus complexes comme la structure de la base de données ou la composition des containers docker.

## Neo4j

Neo4j s'est montré être une solution très adaptée au problème de recherche d'itinéraire. Les requêtes se font très rapidement et les itinéraires retournés semblent corrects. L'apprentissage du langage de requêtes Cypher a été très ludique, grâce la console web qui permet de taper des requêtes et d'afficher leurs résultats de manière graphique ou tabulaire, ou encore de les exporter dans divers formats. Cela a permis d'explorer les possibilités de la base et de prototyper la structure des données très tôt dans le projet.

## Retour sur l'application

L'application délivrée à la fin du travail de semestre n'est finalement qu'une esquisse, un prototype. Elle n'est pas encore prête pour être déployée en ligne car les configurations pour la production n'ont pas été créées, par faute de temps.

Le site n'est pour l'instant pas très sécurisé: par exemple, l'accès aux l'interfaces web de neo4j et de flower ne demandent actuellement aucune authentification. C'est évidemment inacceptable pour une mise en production.

De plus, la recherche d'itinéraire est encore très rudimentaire et mériterait d'être améliorée afin de pouvoir rivaliser avec les autres outils similaires. Il n'est par exemple pas encore possible de calculer des itinéraires dont certaines étapes sont un trajet à pied entre deux arrêts proches, comme le fait l'outil de recherche d'itinéraire officiel des TPG.

L'interface web est encore très basique et minimale, et pourrait intégrer de nouvelles fonctionnalités, telles que la gestion de comptes utilisateurs qui pourraient ajouter des itinéraires en favoris.

## Intérêt personnel

Bien que le projet ne soit qu'à l'état de prototype, j'ai eu beaucoup de plaisir à découvrir de nouvelles technologies, de nouveaux concepts, de nouvelles manières de raisonner autour d'un problème donné.

Le but avoué de ce projet était d'étudier la faisabilité d'un service de recherche d'itinéraire avec des données mises à jour périodiquement dans un graphe persistant. C'est à mes yeux une réussite, une preuve que c'est possible, que les performances sont très bonnes, et que les technologies choisies cohabitent bien ensemble.

J'ai très envie de continuer de travailler sur ce projet et de le faire évoluer jusqu'à ce qu'il soit assez stable, complet et sécurisé pour être mis en ligne, afin de pour pouvoir en faire profiter les internautes qui utilisent les Transports Publics Genevois.

## Table des figures

1	Vue des itinéraires sur le site des TPG . . . . .	9
2	Vue des itinéraires dans l'application Android Öffi . . . . .	10
3	Vue depuis le site web (version mobile) des CFF . . . . .	12
4	Esquisse de la page <code>home</code> . . . . .	13
5	Mockup de la page <code>find_paths</code> . . . . .	14
6	Mockup de la page <code>find_paths</code> avec les détails d'un chemin . . . . .	14
7	Exemple de graphe avec des relations qui ne sont pas dirigées . . . . .	15
8	Exemple de graphe avec des relations dirigées . . . . .	15
9	Comparaison entre les machines virtuelles et les containers. . . . .	17
10	Exemple de noeud <code>commercial_stop</code> (en bleu), lié à des <code>physical_stop</code> (en vert) et des <code>line</code> (en rouge) . . . . .	19
11	Exemple de noeud <code>physical_stop</code> (en vert), lié à des <code>route_step</code> (en gris) et un <code>commercial_stop</code> (en bleu) . . . . .	20
12	Exemple de noeud <code>line</code> (en rouge), lié à des <code>route_step</code> (en gris), des <code>destination</code> (en rose) et des <code>commercial_stop</code> (en bleu) . . . . .	21
13	Aperçu du rendu de la vue <code>home</code> . . . . .	26
14	Aperçu du rendu de la vue <code>find_paths</code> . . . . .	27