Développements d'Applications Réticulaires

Projet Transilien

L'application web à ne pas manquer

COQUART Kevin, NADARAJAH Mag-Stellon

Table des matières

Chapitre 1

Manuel utilisateur

L'accès au site web se fait à l'adresse suivante : `http://transilien-upmc.appspot.com`
Le site se compose d'une seule page, les interactions sont dynamiques.

La page affiche une carte contenante des points de différentes couleurs.
Chaque point représente une gare, les gares sont coloriées suivant les standards de la SNCF, les points en noir sont des nœuds du réseau.
Le bandeau de menu permet d'obtenir des informations sur les concepteurs du site.
Il est possible via un système de layer d'afficher le tracé des lignes.

Le menu déroulant de droite possède 3 fonctionnalités :
Calculer un itinéraire : calcul l'itinéraire le plus court (en distance) d'une gare A vers une gare B. Son exécution affiche sur la carte le trajet à suivre et les prochains trains disponibles pour entamer le trajet.
Information sur une gare : centre et zoom la carte sur la gare indiqué et ouvre le pop-up contenant les lignes accessible à la gare.
État de service : affiche un graphique sur l'état de service des trains contenu dans la base

L'auto-complétion permet de faciliter la saisie des gares lors des demandes de l'utilisateur.

Lors de l'appui sur une gare, une info-bulle s'affiche présentant les lignes accessibles à la station et les prochains trains au départ.

Chapitre 2

Introduction

Ce rapport présente les fonctionnalités de l'application transilien-upmc.appspot.com, ce projet s'est déroulé en plusieurs phases, nous avons d'abord réfléchi à ce que l'on pouvait faire avec une API proposant uniquement les prochains départs à une gare.

Nous n'avons pas trouvé d'idées innovantes, à partir de là, nous avions décidé d'avoir un rendu graphique agréable et de proposer le positionnement des gares, l'affichage des trains au départ des gares, le suivi en temps réels des trains ainsi que le calcul des états de service des trains. Au fur et à mesure du projet, il s'est avéré que tous n'étaient pas réalisables.

Nous allons tout d'abord vous présenter les différentes technologies employées durant ce projet, par la suite, nous ferons un listing des fonctionnalités présentes. Nous ferons un arrêt sur les algorithmes mis en place dans cette application et nous finirons par une description des points forts, des points faibles et des axes d'évolution de notre site.

Chapitre 3

Technologies et Architectures mise en place

3.1 Architecture

Notre site est hébergé sur Google App Engine, ce dernier ne propose pas de base de données à proprement parler mais d'un datastore. Le datastore permet de simuler une base de données mais les limites de la version gratuite sont assez vite atteint.
Pour notre projet, cela reste suffisant.

3.2 Technologie

3.2.1 Serveur

Le serveur est implémenté en Java orienté Web, le Framework app engine permet de simplifier la création des servlets en annotant son code pour déclarer les points entrant de l'API. Le Framework objectify, lui aussi fourni par Google, permet de faciliter la gestion de la persistance avec une gestion à haut niveau. L'utilisation des bibliothèques Json.simple et JDom2 permettent respectivement de parser les fichiers Json et XML. Les fichiers Json proviennent du site de la SNCF, ils sont stockés sur le serveur et permettent d'initialiser l'application. Le parseur XML est utilisé pour traiter les retours de l'API de la SNCF, il prend un flux de donnée en entrée et rend en sortie une liste de train.

3.2.2 Client

Le front end de notre projet est un client léger accessible via n'importe quel navigateur web. Notre application web se décompose essentiellement en des aspects statiques et des aspects dynamiques. Dans la suite, nous allons développer ces points.

Aspects Statiques

Les aspects statiques concernent les éléments qui sont définis en « dur » dans notre projet et ne sont donc pas dépendant du contexte.

La structure de nos pages : HTML/CSS/Bootstrap

Notre application web se devait d'être accessible à un grand nombre d'utilisateur. Et de ce fait, on a décidé de le rendre compatible « cross-browser » et sur les plateformes mobiles. C'est pourquoi nous avons utilisé un certain nombre de propriétés de $HTML5/CSS3$ qui permettent de répondre à notre besoin. De plus, nous nous sommes également appuyés sur le Framework *Bootstrap*.
La navigation sur notre application web est rendu possible par uniquement la création de page HTML statique.

Prenons un exemple d'un bout de code $HTML$ de la page d'accueil

```
<div id="dashboard" class="col-md-2">
 <!-- Debut groupe de bloc collapsable -->
 <div class="panel-group" id="accordion">
  <!-- Debut groupe de bloc calculer un intineraire -->
  <div class="panel panel-default">
   <div class="panel-heading">
    <h4 class="panel-title">
   <a data-toggle="collapse" data-parent="#accordion" href="#
      collapseOne"> Calculer un itineraire </a>
  </h4>
   </div>
```

Faisons quelques commentaires sur ce code $HTML$.

```
<div id="dashboard" class="col-md-2">
```

On adopte un système de grille qui permet de diviser l'écran du navigateur web en 12 colonnes. Ce code permet au navigateur client d'associer à notre bloc dashboard une largeur équivalent à $(ScreenSize/12) * 2$.
Cela est rendu possible par l'utilisation des « media queries » qu'apporte $CSS3$ et assure une grande adaptabilité client.

```
<a data-toggle="collapse" data-parent="#accordion" href="#
   collapseOne">
```

On a également utilisé la propriété « data- » qu'apporte *HTML5*.
Cette utilisation permet de construire des tags plus riches et d'inclure les données directement dans les tags.
Cela évite de déporter les données typiquement dans du *Javascript* et donc d'améliorer l'expérience utilisateur.

3.2.3 Aspects Dynamique

La structure du modèle de données : Javascript/Jquery

Nous avons choisi de compartimenter de manière astucieuse l'exécution de nos codes *Javascript*.

Ainsi, en s'inspirant des langages de programmation, on a mis en place un environnement global dans lequel toutes les données nécessaires sont instanciées. Dans cet environnement se trouve l'ensemble des *URI* de l'*API Rest*, des structures de données contenant les gares, les lignes ...
Et dans un second temps, nous avons défini l'ensemble des objets et les fonctions qui vont être utiles à notre application web.

Nous nous sommes rendus compte, au vu du nombre de ligne écrit, qu'il est difficile de déboguer et de maintenir notre code *Javascript*. On a donc rajouté un système de vérification des types des paramètres formel sur les fonctions et les objets critiques que l'on manipule. Cela a amélioré la productivité du développement de nos codes *Javascript*.

```
function Gare(nom, codeUIC, longitude, latitude) {
  // Ajout du typage des parametres sinon declenchement d'une
     exception
  if( (typeof nom != "string") || (typeof codeUIC != "number") ||
     (typeof longitude != "number") ||
      (typeof latitude != "number") ){
    lancerException("Les parametres ne correspondent pas au typage
        de la fonction Gare");
  }
```

La carte interactive : Javascript/Jquery/Leaflet

Nous avons eu l'idée d'incorporer une carte interactive à notre application web. Néanmoins, notre critère de sélection était d'avoir complètement la main sur la carte interactive choisie. On a donc écarté *GoogleMaps* et *MapBox* pour se tourner vers la solution *Leaflet* : une carte interactive, entièrement développée en *Javascript*, très légère, ultra configurable et open source.

Afin de positionner l'ensemble des gares, il suffit de faire une requête *Ajax* à notre back end qui nous renvoie une liste de gares associée à leur coordonnée géographique. Ensuite, à l'aide de l'*API* de *Leaflet*, on manipule la carte pour placer les objets Gares sur la carte.

L'interface utilisateur : Javascript/TypeHead/BloodHound

L'une des fonctionnalités de notre application web est de calculer l'itinéraire entre deux gares.
Afin que l'expérience utilisateur soit améliorée, nous avons mis en œuvre la recherche des gares de départ et de destination par un système d'auto-complétion.

Nous nous sommes donc appuyé sur *TypeHead* qui nous offre une interface graphique « user friendly » pour l'auto-complétion. Concernant la partie logique, notre première implantation fut la suivante :

```
var findSubString = function(strs) {
  return function findMatches(q, cb) {

    var matches, substrRegex;
    matches = [];
    substrRegex = new RegExp(q, 'i');

   $.each(strs, function(i, str) {
      if (substrRegex.test(str)) {
        matches.push({ value: str });
      }
    });

    cb(matches);
  };
};
```

On remarque très rapidement que ce type d'algorithme est assez coûteux. En effet, cette fonction est appelée à chaque fois que l'utilisateur tape un caractère dans un des champs d'auto-complétion (*Jquery* KeyboardEvent bindé sur le champ input). Et, on craignait donc des latences de réponses notamment sur les plateformes mobiles.

Nous nous sommes donc appuyés sur les fonctionnalités de *Bloodhound*. Nous avons donc pu améliorer les performances de l'auto-complétion par des techniques de caches astucieuses, du chargement des informations des gares en anticipant la saisie de client et d'algorithmes de recherches optimisés.

Chapitre 4

Fonctionnalités

Il est possible comme nous l'avons évoqué dans le manuel de l'utilisateur de réaliser plusieurs actions sur le site :

- Accéder à une carte avec le schéma du réseau (gare + tracé des lignes)
- Visualiser les lignes accessibles à une gare
- Visualiser les trains au départ d'une gare
- Calculer un itinéraire d'une gare A à une gare B et obtenir les prochains trains le permettant
- Consulter un graphique représentant l'état de service des trains de la SNCF (vu le faible jeu de donnée que nous possédons, le graphique indique un taux de service de 100

Ceci représente les fonctionnalités visibles sur le client. Des fonctionnalités essentielles à l'application :

- Initialiser l'application
- Nettoyer le dataStore (pour ne pas outre passer le quota)
- Récupérer toutes les gares
- Récupérer une gare par son id / son nom / son(ses) codeUIC
- Récupérer le nom de toutes les gares (pour l'auto-complétion)
- Récupérer les lignes
- Récupérer les gares par ligne

- Récupérer les lignes d'une gare
- Récupérer tous les trains

En dehors des fonctionnalités accessible via l'API, il y a plusieurs algorithmes qui tournent.

Chapitre 5

Algorithme

5.1 Initialisation

L'initialisation est en plusieurs phases, nous chargeons tout d'abord les gares depuis un fichier Json télécharger sur le site de la SNCF, par la suite, nous chargeons les lignes depuis un Json provenant du même endroit.
À ce stade, certaines gares ne correspondant à aucune ligne et sont donc obsolète, c'est pour cela que nous filtrons les gares de la base via celle appartenant à une ligne.
Par la suite et pour être en mesure de tracer le schéma du réseau, on oriente (enfin on essaie [voir plus loin la description de l'algo]) les lignes dans le but de sauvegarder la liste de gares dans l'ordre et de ce fait être en mesure côté client de tracer le plan des lignes.

```
public void filtreGareNonTransilienne() {
    List<Gare> gares = allGare();
    List<Ligne> lignes = allLigne();

    List<Integer> existInLine = new List();
    for(Ligne l : lignes)
        existInLine.addAll(l.getGare());

    for(Gare gare : gares)
        if(!existInLine.contains(gare.codeUIC()))
            GareRepo.remove(gare);
}
```

5.2 Recherche d'un itinéraire

L'API de la SNCF propose les prochains départs des trains sur une même ligne, il n'est pas rare de devoir effectuer des changements pour se rendre d'un point A à un point B.
Nous avons voulu régler par nous-même ce problème.

Pour ce faire, nous appliquons le principe de dijkstra.
Tout d'abord, nous vérifions si un direct existe, si oui, c'est fini.
Sinon, la gare de départ est affectée d'une distance de 0, toutes les gares accessibles directement via le départ sont ajoutées à une file de priorité suivant leur éloignement au départ.
Pour chaque gare de la file de priorité, nous itérons cette phase.
Lorsque la gare d'arrivée est affectée d'une distance, nous regardons la 1ère valeur de la file de priorité, si elle est supérieure à celle de l'arrivée ou bien si la file est vide, nous avons terminé.
Dans le cas contraire, on continue d'itérer dans le but d'améliorer la distance est d'être sûr de rendre le plus court chemin.

Une fois ceci fait, en partant de la gare d'arrivée, on trace l'itinéraire grâce au prédécesseur (calculé en même temps que la distance) de chaque gare.

```
GareDijk {
    Gare gare;
    Double distance;
}

public List<Gare> itineraire(Gare a, Gare b) {
/* 1) Recherche d'un directe */
    if(direct(a, b))         return [a, b];

    Map<GareDijk, GareDijk> predecesseur;
    FileDePriorite<GareDijk> tas;
    tas.add(new GareDijk(a, 0));

  /* 2) Parcours de toutes les gares tant que le tas n'est pas
     vide et que la ditance pour aller a la destination et
     superieur a la distance de la 1er gare du tas */
    while(!tas.isEmpty() && distances.(b) > tas.peek().distances)
       {
        /* 2.1) On itere sur les lignes qui passent par la gare */
        for (Ligne ligne : LigneRepo.findLigneByGare(a))

            /* 2.2) On itere sur les gares de chaque lignes */
            for(Gare g : ligne.gares()) {
                /* On calcul la distance, si elle ameliore le
                   score, on met a jour le score et le
                   predecesseur. Puis on actualise la position
                   dans la file de priorite */
            }
    }
```

```
    /* 3) On part de la destination pour creer la liste de gare
       via le predecesseur. */
    List<Gare> gares;
    GareDijk tmp = b;
    while(tmp != null) { /* ... */ }
    return gares;
}
```

Le fait de calculer pour chaque ligne et non pas seulement pour les voisins alourdit l'algorithme, mais nous ne possédons pas les voisins, et malgré notre algorithme d'orientation des lignes, nous n'obtenons pas un résultat exploitable en terme de programmation (tout de même suffisant pour de l'affichage). De plus, de ce fait, nous obtenons un itinéraire contenant uniquement les correspondances et aucune gare intermédiaire.

5.3 Orientation des lignes

Dans le but d'afficher le tracé des lignes, nous devions orienter les lignes pour avoir une liste de gares dans l'ordre dans lequel elles apparaissent.

Les deux plus proches voisins d'une gare ne sont pas toujours ces deux prédécesseurs.
Pour pallier ce problème, nous considérons que la gare la plus proche est forcément son voisin, pour trouver son 2ème voisin, nous prenons la gare la plus proche qui admet un angle directionnel (par rapport au 1er voisin) supérieur à 45°. Nous avons estimé que c'était une mesure suffisante pour éviter de prendre les deux plus proches voisins qui serait du même côté de la ligne.

Les lignes de la SNCF ont un autre souci majeur, elles sont constituées de plusieurs branches. Comment détecter les changements de branche ? Nous ne pouvons pas, pour éviter d'amener trop d'erreurs sur le calcul du tracé, nous considérons que les deux plus proches voisins sont dans un cercle de 15 km. Dépassé cette distance, on estime que ce n'est sûrement plus la même branche.

```
Couple {
    String nom;
    Double distance;
}

Voisin {
    String voisin1;
    String voisin2;
}

public static List<Gare> oriente(Ligne ligne) {
```

```
/* 1) On ajoute a une map toutes les gares de la ligne */
    Map<String, Gare> mapGare;
    List<String> aTraiter;
    for(Gare g : ligne.getGare()) { /* ajout */ }

/* 2) On associe a chaque gare un Voisin */
    Map<String, Voisin> voisins;
    voisins.put(allGare.nom, new Voisin());

/* On traite chaque gare de la ligne */
    while(!aTraiter.isEmpty()) {

        /* 3) Creation d'une file de priorite base sur des couples
            <codeUIC, distance / la gare traite> */
      FileDePriorite<Couple>  distances;

        /* 4) Calcule des distances de toutes les gares non traite
            / a la gare de ref (1er gare de la liste aTraiter) */*
      for(String nom : aTraiter) { /* maj distances */ }

        /* 5) Tant que la file n'est pas vide, on recherche les
           voisins pour les ajouter.<br>
 Il ne faut pas non plus que la gare la plus proche soit a plus de
     20 km, sinon c'est surement que l'on traite le bout d'une
    ligne.<br>
De meme, si les voisins sont remplis, on la retire des gares a
   traiter.*/
    while (!distances.isEmpty() && (distances.peek().distance <
       20000) && (voisin.voisin1 == null || voisin.voisin2 == null
       )) {

            /* 5.1) Le voisin1 est null, le voisin2 aussi par
               construction.<br>
On indique aux gares qu'elles sont voisines.*/
            if(...) { ... }

            /* 5.2) Le voisin2 est null, sinon c'est qu'il y a un
               soucis de construction.<br>
On calcul l'angle entre le voisin1 et le potentiel voisin2, si c'
   est superieur a 45 deg, c'est l'autre partie de la ligne. On
   indique aux gares qu'elles sont voisines. */
            else { ... }
    }
    }
    return goodOrder(mapGare, voisins);
```

```
}


private static List<Gare> goodOrder(Map<String, Gare> mapGare, Map
   <String, Voisin> voisins) throws Exception {
  List<Gare> gares;
  List<String> aTraiter = new List(mapGare.keySet());

  // Tant qu'il y a des gares
  while (!aTraiter.isEmpty()) {
    // On recherche un terminus de la ligne
    debut = /* terminus trouve */

    /* On ajoute les gares a la liste dans l'ordre via les voisins
        calcule plus-tot */
    while (debut != null) {
            /* On ajoute la gare, on part sur le bon voisin, etc
               ... */
    }

    /* On ajoute un null a la liste, il sert d'intercalaire entre
       2 groupes de gare oriente. */
    gares.add(null);
  }
  return gares;
}
```

Globalement cette algo permet d'avoir une idée du tracé des lignes mais n'est pas du tout exploitable pour calculer ou pour indiquer le tracé exacte d'une ligne.
Pour être en mesure de tracer les lignes comme il faut, il aura fallu modifier le fichier Json des lignes pour les orienter à la main.
Nous avons préféré avoir un rendu moins précis mais plus osé, le choix de rester sur un algorithme pour permettre une évolution des lignes (dans le cas d'une manipulation manuelle, que fait-on si des lignes sont modifiées (travaux, fermeture de station, etc...)), de ce fait, nous perdons en précision, mais conservons un challenge qui aura été en partie résolu.

5.4 Tache Cron

Une tâche automatisée se charge de remplir la base de train (pour améliorer la finesse des statistiques), pour éviter d'avoir un trop grand nombre d'appels à la base, les requêtes sont fixées.
Nous ajoutons à notre base l'ensemble des trains au départ de Châtelet, Gare de Lyon et la Défense toutes les 30 minutes.
Les gares ont été choisies en fonction de leur fréquentation, ce sont des gares importantes et

possédant quelques lignes transiliennes.

Chapitre 6

Point fort

6.1 Évolutivité

L'initialisation se fait via des fichiers Json récupérés depuis le site de la SNCF, cela permet si le réseau change de télécharger le dernier Json, de l'inclure au serveur et de réinitialiser l'application.
Cette suite d'opérations permet d'être réactive, le traitement étant automatisé, nous restons sur évolution du réseau rapide, il n'est pas nécessaire de traiter les données à la main pour les ajouter.

6.2 Robustesse

L'utilisateur n'accède pas directement aux requêtes, quelle soit sur l'API SNCF ou sur le serveur, de ce fait notre application résiste aux différentes failles xss.
Les requêtes de l'utilisateur, par exemple les codes UIC, sont vérifié côté serveur pour éviter de faire des demandes sur des numéros qui n'existerait pas. Dans le cas d'une demande sur un code inexistant, l'application ne réagit pas.

6.3 Interface immersive

Le client affiche une carte qui remplit toute la page avec les gares présentes.
L'utilisateur est directement dans le schéma du réseau, il aperçoit tout de suite les différentes gares ainsi que le menu lui donnant l'accès aux fonctionnalités de l'application.

6.4 Interface intuitive

Les champs demandés pour interagir avec l'application disposent de l'auto-complétion, il devient donc très facile d'effectuer des recherches simplement, sans se tromper dans le nom et ainsi éviter les erreurs.
Le passage de la souris sur point affiche immédiatement le nom de la station, un clic affiche une info-bulle contenant les prochains départs, tout ceci se fait dans la plus grande transparence.

Chapitre 7

Point faible

7.1 GAE en version gratuite

Google App Engine limite le nombre de requêtes possible ainsi que la taille du dataStore, des limites que nous atteignons très rapidement.
De plus, les protocoles d'échange avec une API externe ou pour l'envoi des mails est restrictifs, il existe très peu de protocoles supportés par la version gratuite.

7.2 Envoi de mail

Le code existe, le test en local indique bien qu'un mail devrait être envoyé, lors du déploiement, les mails envoyés sont comptabilisés dans le quota, mais la réception se fait attendre. De ce fait, nous n'avons pas réussi à envoyer un mail de rappel contenant l'itinéraire et les horaires des prochains trains pour le réaliser.

7.3 Tracé des lignes

Comme nous l'avons évoqué plusieurs fois aux cours de ce rapport, le challenge d'orienter les lignes n'a été rempli que partiellement, le fait que les lignes possèdent plusieurs branches a posé soucis.
Cela implique un tracé approximatif, pour ne pas avoir un rendu brouillon, l'affichage des lignes se fait via un gestionnaire de layer qui permet d'afficher au choix une sélection de ligne, aucune ou bien toutes les lignes.
Grâce à cela, nous pourrions presque dire que cette fonctionnalité n'est qu'une approximation et ne pas la considérer comme un point faible.

7.4 Calcule des itinéraires

Nous en avons également parlé précédemment, le fait de calculer la distance avec l'intégralité des gares présentes sur les lignes accessibles augmente le temps de latence de l'algorithme. Nous n'avons pas soumis l'application à des tests de charge, mais dans une utilisation de grande envergure, le temps de calcul pourrait poser problème.
Il n'est actuellement que de quelques secondes, mais qu'en serait-il avec des milliers d'utilisateurs.

7.5 Échec face à l'application de la RATP

Notre application bien que d'un esthétisme bien supérieur à celle de la RATP, et l'absence de pub pour Disney, n'égale pas celle-ci en terme de fonctionnalités.
Leur fond de carte statique est bien plus rapide à charger que la nôtre qui est dynamique.
Leur tracé des lignes est impeccable, tandis que le nôtre est approximatif.
Leur calcul d'itinéraire indique le temps de déplacement et prend en compte un nombre de transport plus important que le nôtre.
Malgré tout cela, qu'en aurait-il été s'il avait dû développer leur application avec les ressources mise à disposition via la SNCF et en effectuant uniquement des traitements automatiques ?

Nous avons tout de même l'affichage des statistiques sur le taux de service des trains, bien que partiel, la RATP préfère cacher cette donnée pour ne pas décourager ces utilisateurs.

Chapitre 8

Axe d'évolution

8.1 Débloquer le quota

Le quota limite le nombre de requêtes en lecture et en écriture vers la base, notre application atteint les limites du quota très rapidement, dans le cadre d'un lancement, il faudrait le débloquer en achetant des ressources chez Google.

8.2 Réaliser des tests

Nous n'avons automatisé aucun test sur l'application, la vérification des fonctionnalités et visuel ou bien par l'absence de bug sur nos tests manuel (que se soit via un accès direct à l'API ou via le client).
Réaliser des cas de tests permettrait de s'assure du bon fonctionnement de l'ensemble et d'assuré une non-régression des fonctionnalités lors du déploiement d'amélioration.

Nous aurions deux ensembles de tests, des tests unitaires dans le but de vérifier le fonctionnement de chaque fonction du serveur et à côté un ensemble de tests d'intégration pour s'assurer du bon fonctionnement de l'application et des échanges avec le client Web.

Chapitre 9

Conclusion

Ce n'est pas la 1ère fois que nous réalisons une application Web, notre connaissance des technologies Web (HTML / CSS / Js) ainsi que de J2EE est bonne.

Néanmoins, ce projet nous a permis d'utiliser des Framework ou d'essayer des bibliothèques que nous ne connaissions pas (GAE, LeatFlet, ...).
L'échéance relativement courte du projet était un défi, ils nous fallait estimer le temps de développement de chaque fonctionnalité pour rester dans le délai imparti et livrer lors de la soumission finale un produit fonctionnel.

Le découpage du travail ce fit facilement et plutôt équitablement, ce qui est suffisamment rare à l'UPMC pour être soulignée. Il y a eu de nombreux échanges entre la partie développement Front et Back, cela nous a permis d'assembler les 2 sans trop de problèmes.

Pour conclure, même si ce projet ne nous a pas apportés de nouvelles connaissances, il a tout de même mis en avant la validation d'acquis ainsi que la tenue d'un planning.