Documentation technique

Projet « Train Commander » Supinfo 2015 – 2016

Sommaire

[Choix des technologies 2](#_Toc452749194)

[Nos outils 2](#_Toc452749195)

[Fonctionnalités du site 4](#_Toc452749196)

[Côté utilisateur 4](#_Toc452749197)

[Côté administration 4](#_Toc452749198)

[Explication de l’algorithme 5](#_Toc452749199)

[Principe 5](#_Toc452749200)

[Recherche par temps 6](#_Toc452749201)

[Recherche par coût 6](#_Toc452749202)

[Gestion des correspondances 7](#_Toc452749203)

[Diagramme de classe 7](#_Toc452749204)

[Optimisation de la base de données 8](#_Toc452749205)

[Schéma de données 8](#_Toc452749206)

[Stockage des utilisateurs et d’informations spécifiques 9](#_Toc452749207)

[Stockage des gares 9](#_Toc452749208)

[Stockage des trajets 10](#_Toc452749209)

[Fonctionnalités bonus 11](#_Toc452749210)

[Application Windows 11](#_Toc452749211)

[Web API 11](#_Toc452749212)

[Back office du site 12](#_Toc452749213)

# 

# Choix des technologies

En ce qui concerne le site web, nous avons choisi d’utiliser le framework PHP **Symfony2**. Puissant et très répandu, Symfony est un choix logique si l’on considère notre objectif : créer un site web rapide et robuste, capable de renseigner des milliers d’utilisateurs chaque jour. Symfony est également *scalable*, comporte de nombreux modules (mailer, gestion des utilisateurs, etc.) et nous a même permis de créer notre propre **back office**.

Pour ce qui est des applications mobiles (smartphones, tablettes), nous avons mis à contribution les compétences acquises au cours de l’année en les développant en langage natif.

* Java pour Android ;
* C# pour Windows Phone ;
* Swift pour iOS.

Nous avons ainsi obtenu des applications adaptées à leur support.

Enfin, la **Web API** (une de nos fonctionnalités bonus) a été écrite en PHP. Elle sert de liaison entre la base de données et le site web, ainsi qu’avec les applications mobiles.

## Nos outils

Nous avons utilisé :

* PHPStorm pour le site web et la Web API ;
* Android Studio pour l’application Android ;
* Visual Studio pour les applications Windows et Windows Phone ;
* XCode pour l’application iOS.

# 

# Fonctionnalités du site

## Côté utilisateur

* Créer un compte via formulaire.
* Créer un compte par Facebook.
* Créer un compte par Google.
* Se connecter (combo email / mot de passe).
* Se connecter par son compte Facebook ou Google.
* Se déconnecter.
* Chercher des trajets.
  + Les plus rapides.
  + Les moins chers.
* Acheter des trajets (paiement par Paypal).
* Accéder et rectifier les informations de son profil.
* Revoir ses trajets.
* Imprimer ses trajets en PDF.
* Rechercher un trajet à partir d’un trajet déjà effectué.
* Supprimer son compte.

## Côté administration

* Voir les informations du serveur.
* Gérer l’ensemble des employés (voir, ajouter, éditer, supprimer).
* Mettre le site en maintenance.
* Voir l’ensemble des clients.
  + Accéder à leurs informations.
  + Les désactiver.
* Gérer l’ensemble des commandes effectuées (voir, supprimer).
* Gérer l’ensemble des gares (voir, ajouter, éditer, supprimer).
* Gérer l’ensemble des trajets (voir, ajouter, supprimer).
* Gérer l’ensemble des segments (voir, ajouter, éditer, supprimer).
* Gérer l’ensemble des zones (voir, ajouter, éditer, supprimer).

# Explication de l’algorithme

## Principe

On définit par **« meilleur trajet »** le trajet correspondant le mieux aux critères de l’utilisateur : le moins cher ou le plus rapide.

L’algorithme de recherche doit répondre à la question suivante : parmi tous les trajets, comment trouver le meilleur en fonction des critères de l’utilisateur ?

Nous avons divisé cette question en deux sous-problèmes.

1. Parmi tous les trajets imaginables, comment obtenir uniquement ceux qui partent de la station demandée et arrivent à la bonne station de la manière la plus optimisée ?
2. Parmi les trajets ainsi obtenus, comment choisir ceux qui correspondent le mieux aux critères de l’utilisateur ?

Dans un souci de performances et afin d’éviter des calculs inutiles, nous avons décidé que l’algorithme travaillerait sur des zones définies. Cela permet que si un utilisateur cherche un voyage en zone parisienne, l’algorithme ne travaille qu’avec les trajets présents en zone parisienne. Chercher une solution à Brest ou Marseille n’a aucun sens dans ce cas-là.

Dans le cas où l’utilisateur voyage à travers plusieurs zones, on travaille avec les trajets des zones concernées.

Lorsque l’utilisateur cherche à aller d’une zone vers une autre, on va d’abord chercher le meilleur trajet pour le faire sortir de sa zone de départ. On va donc favoriser la recherche de trajets **interzones**, c’est-à-dire de trajets qui traversent plusieurs zones, ainsi que les trajets passant par la **capitale** de la zone.

Une capitale de zone est une ville possédant une gare d’importance majeure à l’échelle locale. Elle est donc plus susceptible que toute autre gare d’avoir accès à des trajets menant vers d’autres zones. D’où l’intérêt de favoriser la recherche de correspondances dans ces gares lors de trajets traversant plusieurs zones.

Le meilleur trajet est donc composé d’une succession optimisée de sous-trajets.

La liste des meilleurs trajets est obtenue en variant les paramètres de recherche : temps de départ variable autour du temps entré par l’utilisateur, 2ème meilleur trajet, etc.

## Recherche par temps

Pour trouver les trajets les plus rapides, nous utilisons le **Connection Scan Algorithm** (CSA), un algorithme bien plus récent et surtout plus performant que celui de Dijkstra, traditionnellement utilisé en théorie des graphes.

Le CSA travaille avec des **« connexions »** qui représentent un train allant d’une gare à une autre à un temps donné. Un trajet correspond alors à une suite de connexions. Les connexions sont d’abord triées par heure de départ croissante avant d’être traitées par l’algorithme.

Le CSA va ensuite définir le trajet le plus rapide pour arriver d’une gare A à une gare B[[1]](#footnote-1).

## Recherche par coût

Malheureusement, le Connection Scan Algorithm n’est pas compatible avec une recherche par coût. Son manque de flexibilité est le prix à payer pour sa puissance.

Nous avons donc mis en place un algorithme plus classique que le CSA pour la recherche des trajets les moins chers. Il répond séparément aux deux questions posées en première partie.

Pour trouver les trajets menant à la station d’arrivée, nous utilisons une fonction **récursive**. D’abord, nous obtenons les connexions correspondant aux trajets possibles d’après les critères de lieu et de temps choisis par l’utilisateur. Autrement dit, ce sont les trajets dans une zone géographique donnée à partir du temps de départ voulu.

Puis, à partir de la gare de départ, on cherche toutes les gares atteignables avec les connexions disponibles. Pour chaque gare ainsi trouvée, on cherche les gares atteignables avec le reste des connexions à notre disposition, etc. Jusqu’à arriver à la gare de destination.

Comme nous limitons le nombre de gares et de connexions grâce à des zones géographiques et des limites temporelles, la charge de travail est relativement faible pour de la récursivité. De plus, nous avons pris des dispositions pour éviter les surcoûts de travail inutiles et pour éviter à l’algorithme de tourner en rond. Par exemple, les gares ne peuvent être visitées qu’une seule fois.

L’algorithme étant conscient du trajet effectué, il donne la liste des trajets physiquement possibles à la fin des fonctions récursives. Nous n’avons plus qu’à trier ces trajets par prix pour obtenir le moins cher. En cas d’égalité, des sous-critères tels que le temps total ou la proximité avec l’heure de départ originale entrent en compte.

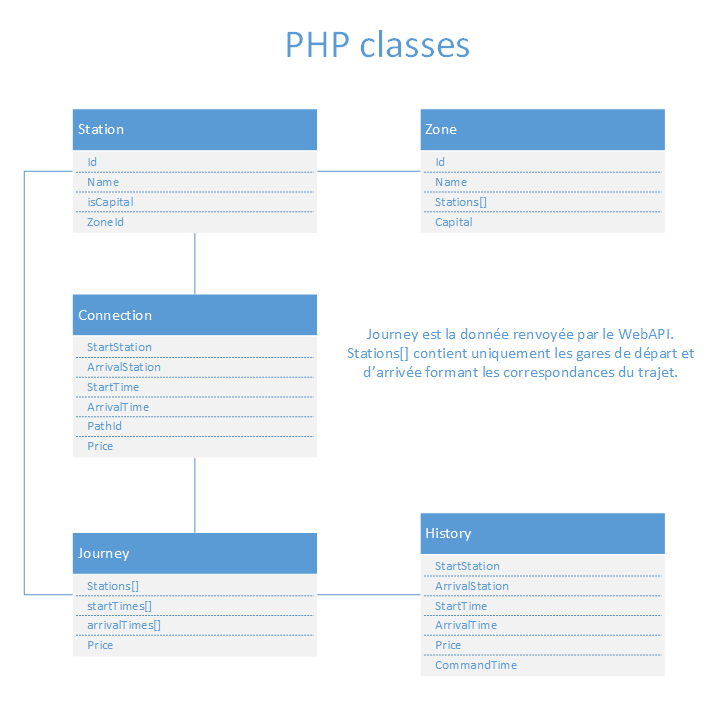
## Gestion des correspondances

Nous nous sommes posé beaucoup de questions quant à la façon dont nous allions gérer les correspondances. La réponse nous ait finalement venue des connexions.

Nous avons dit qu’un trajet était une succession de connexions. Comme l’algorithme ne travaille qu’avec des connexions, il n’a pas besoin de savoir à quel train correspond telle ou telle connexion ! En revanche, l’information lui est disponible.

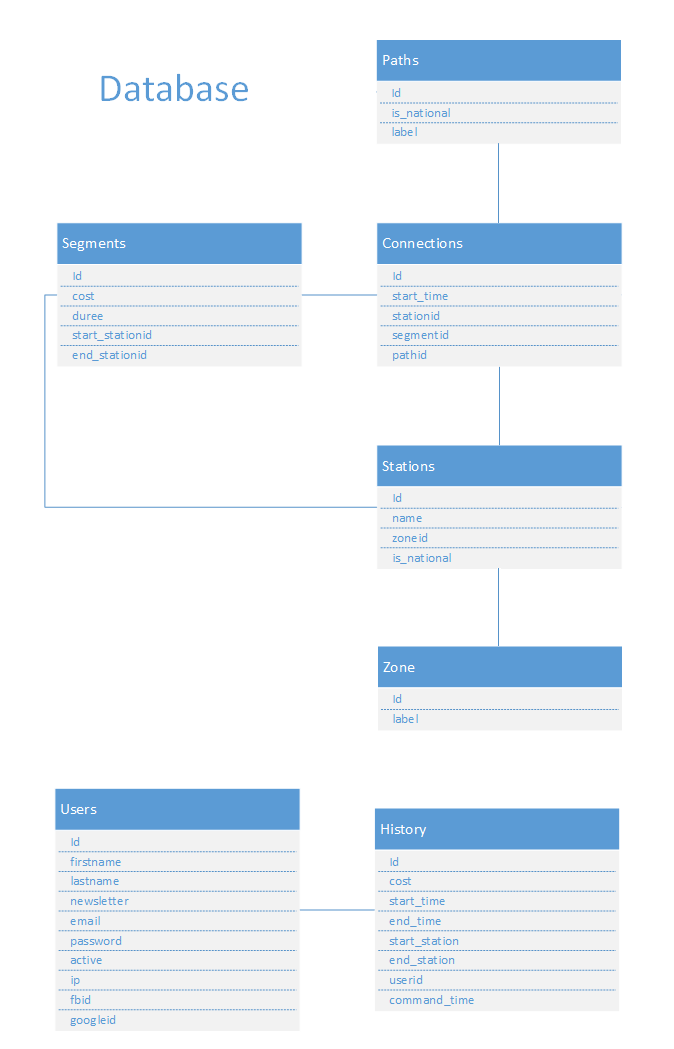
Lorsque le trajet final est déterminé, nous n’avons plus qu’à regarder à quel train appartient chaque connexion pour déterminer les changements de train.

## Diagramme de classe



# Optimisation de la base de données

## Schéma de données



## Stockage des utilisateurs et d’informations spécifiques

Comme vous pouvez le constater sur le schéma ci-dessus, la table qui contient les utilisateurs est plutôt **classique** : informations personnelles, mot de passe crypté, champ permettant de désactiver le compte, informations relatives aux réseaux sociaux.

La table History contient toutes les commandes passées sur le site. Chaque commande est liée à un utilisateur. Nous préférons enregistrer les commandes dans une table à part pour nous assurer que ce que l’utilisateur commande reste constant. Nous verrons en effet par la suite que certaines données sont calculées de manière dynamique. Stocker en dur évite également de devoir recalculer ces données dynamiques à chaque consultation de l’historique.

En somme, nous prenons un peu plus de place pour gagner en performances.

Certaines tables ne sont pas montrées sur le schéma, car elles ne sont pas le résultat de choix majeurs dans la construction de notre base de données. Par exemple, nous avons une table pour les employés et une table réservée à divers éléments de configuration du site web.

## Stockage des gares

Cette section et la suivante vont mettre en lumière un de nos objectifs principaux lors de la conception de la base de données : supporter au maximum l’implémentation des algorithmes de recherche.

Nous avons dit dans la section dédiée à l’algorithme que nous avions découpé le réseau ferroviaire en zones. Cela se retrouve dans la base de données avec les tables Stations et Zone. Le bénéfice direct de cette organisation est que les requêtes sur les gares sont ciblées. Comme on ne cherche que des gares par rapport à une zone, les requêtes sont rapidement exécutées et les résultats retournés sont légers.

Mieux encore, à chaque zone, sa capitale. Un champ permet directement d’identifier les gares importantes de chaque zone afin d’orienter les trajets vers elles.

Quant aux segments, ils représentent les liaisons physiques (les rails) entre les gares. Ils n’ont pas de sens, possèdent un coût d’exploitation ainsi que le temps nécessaire pour rallier la gare 1 à la gare 2. Ensembles, ils forment le réseau ferroviaire.

## Stockage des trajets

Nous avons défini un trajet comme un train allant d’une gare A à une gare Z terminus à un temps donné. Un trajet peut donc être décomposé en sous-trajets (A → B, B → C, C → … Z) appelés connexions.

On définit alors une connexion comme un train allant d’une gare A à une gare B. Contrairement aux segments, les connexions ont un sens, commencent à un temps donné et sont à usage unique.

Nous avons dit que le CSA a besoin de connexions pour fonctionner. Ces connexions correspondent à une classe PHP. Or, les informations nécessaires à créer une connexion (classe PHP) sont réparties dans la table Connections et dans la table Segments. Pourquoi avoir séparé les informations ?

D’une part, pour gagner de l’espace de **stockage**. Comme chaque trajet engendre des connexions, on se retrouverait à stocker une grande quantité de gares, de prix et de temps souvent similaires. Si les connexions ne contiennent plus que les informations dynamiques (sens, heure de départ) en laissant les informations statiques (prix, temps de trajet, gares) aux segments, on gagne beaucoup d’espace de stockage sur les connexions.

D’autre part, pour avoir une **modélisation logique** des éléments. Les informations des segments ne sont pas censées changer, tout comme les rails posés sur le sol. Si le coût d’un segment augmente, tous les trajets passant par ce segment coûteront plus cher.

D’un point de vue mathématique, les segments sont des droites, des figures sur lesquelles on peut appliquer des vecteurs, qui sont nos connexions.

La base de données reflète ainsi la réalité en se basant sur un modèle **intuitif et optimisé**.

Autre exemple d’optimisation : la différenciation des trajets interzones. Nous avons dit que l’algorithme allait chercher des trajets interzones si l’utilisateur a besoin de traverser plusieurs zones. La base de données peut directement les lui fournir !

En effet, les trajets ont un flag is\_national qui, mis à la création, indique si le trajet traverse plusieurs zones ou non. Nous aurions faire une sous-requête vérifiant chaque gare du trajet à la place, mais cela aurait été une charge supplémentaire.

On gagne ainsi beaucoup de performances à l’aide d’un simple booléen.

# Fonctionnalités bonus

## Application Windows

Lors du développement de l’application mobile pour Windows Phone, nous avons choisi de faire une application universelle. Les applications universelles permettent d’avoir une même application à la fois compatible avec Windows Phone et Windows.

Elles demandent plus de travail puisqu’il faut que l’application puisse s’adapter à plus de supports, mais nous avons estimé que cela valait le coup.

Nous pensons que cela a un intérêt notamment auprès personnes peu habituées à l’outil informatique et des seniors. Plutôt que d’avoir à lancer un navigateur internet et faire une recherche pour aller sur le site, il suffit de lancer l’application pour avoir accès à tous nos services, que ce soit sur mobile ou sur un ordinateur avec Windows.

## Web API

Comme expliqué en introduction, la Web API est l’élément permettant de faire le lien entre toutes les parties impliquées dans notre écosystème : site web, applications mobiles, base de données. Vous trouverez un manuel d’utilisation joint à ce document.

C’est à partir d’elle qu’on accède à la base de données, et c’est elle qui s’occupe de chercher les trains via les algorithmes présentés plus haut.

Elle permet :

* **Un meilleur contrôle de l’accès aux données**. En effet, l’utilisateur ne demande jamais rien à la base de données directement. Il doit passer par la Web API, d’où une meilleure sécurité.
* **Un accès uniformisé aux données**. On évite par exemple de devoir installer une base de données locale sur le smartphone d’un utilisateur, ce qui prendrait de la place et offrirait des possibilités de piratage.
* **De faire de l’Open Data**. Nous suivons la tendance de la SNCF notamment. Nous permettons à n’importe qui d’accéder à nos données (de manière contrôlée, bien entendu) et à notre algorithme de recherche de trains. Nous encourageons ainsi les développeurs à créer des applications et à promouvoir nos services.

## Back office du site

Vous avez pu voir dans la liste des fonctionnalités du site que nous permettons aux administrateurs de gérer de nombreux aspects du site. Nous avons en effet trouvé dommage de créer un site web moderne sans autres possibilités pour le gérer que d’aller dans le code ou de modifier les valeurs en dur dans la base de données.

C’est pourquoi nous avons développé un back office qui, s’il n’offre pas toutes les fonctionnalités d’un véritable CMS, s’en approche néanmoins.

Avec la possibilité d’accéder à toutes les informations utiles du serveur, de mettre le site en maintenance facilement, de gérer les données stockées avec aisance (utilisateurs, trajets, commandes, etc.), nous pensons apporter une **importante plus-value** à notre site.

Vous trouverez avec cette documentation un manuel de déploiement ainsi que qu’un manuel d’utilisation pour la Web API.

1. Plus d’informations sur le fonctionnement du CSA : http://www.dericbourg.net/2015/12/10/calcul-ditineraire-a-partir-des-donnees-ratp/ [↑](#footnote-ref-1)