Projet « OPTRAJ »

Documentation technique

Thomas Bouguet, Nicolas Carre, Guillaume Chapel,

Corentin Clément, Alban Drézen, Florian Etourneau,

Vivien Lelouette, Nicolas Poirey, Axel Verax,

Benjamin Soulas, Jérémy Vigneron

Supervision : Guillaume Pierre

Sommaire

Introduction

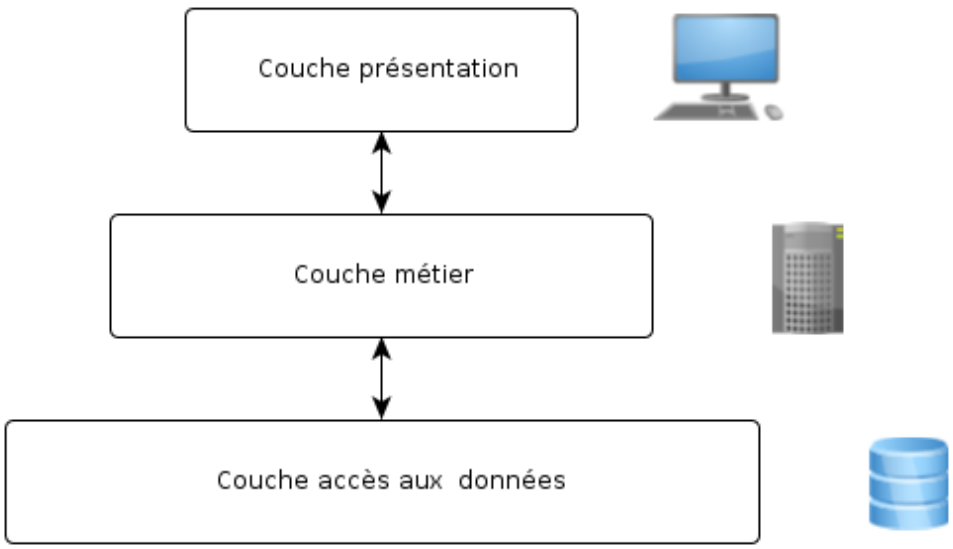
L'architecture du logiciel OPTRAJ

Le logiciel OPTRAJ est organisé en trois couches aussi appelées architecture **trois-tiers** :

La couche présentation qui est la couche la plus haute du système. Comme son nom l'indique, cette couche représente l'interface utilisateur du logiciel. Elle est codée en **HTML5/CSS**, **JavaScript** ainsi que quelques fichiers **PHP**.

Vient ensuite la couche métier qui regroupe toute l'algorithmique du projet, les classes Système ainsi que les API qui servent à communiquer avec les autres couches du logiciel. Cette partie a été développée en **Python** (v2.7.6).

Enfin, la couche persistante (ou couche base de données) contenant la base de données du logiciel OPTRAJ. Le langage utilisé ici est **MySQL**.



Les outils utilisés

Durant ce projet nous avons utilisé différents outils qui nous ont aidé à la bonne réalisation de ce projet.

Le premier outil utilisé est **GIT**. Nous avons profité d'un hébergement sur la forge de l'ISTIC pour nous permettre d'utiliser ce système de contrôle de version gratuitement tout en pouvant garder les sources secrètes. **GIT** a permis à chaque membre de l'équipe de travailler sur le code en même temps que les autres en étant assuré d'avoir un outil fiable gérant les potentiels conflits d'édition.

Le second outil employé est **Trello**. C'est un outil de gestion de projet en ligne qui nous a permis de mettre en pratique notre méthode de travail **SCRUM**. Cet outil, à la différence d'autres (comme iceScrum que nous utilisions précédemment), est très simple d'accès (application WEB, mobile) et nous a permis de tenir au jour le jour l'avancée de nos travaux.

Le troisième outil utilisé lors du projet est **Vagrant**. Ce logiciel permet de créer et gérer très facilement un environnement de développement virtuel. Le principal avantage de cette solution est qu'une fois l'environnement mise en place, tout le monde peut très facilement booter la machine virtuel sur son propre ordinateur sans avoir à faire toute la configuration spécifique à son système. Pour plus d'informations sur l'installation et l'utilisation de **Vagrant** vous pouvez vous rendre en Annexe 3.

Enfin, nous avons utilisés **Doxygen** avec **Doxypypy** pour la documentation python. En effet, **doxypypy** est un script Python réalisé dans le but de transformer les commentaires compatibles pour **doxygen**. On peut donc garder des commentaires facilement lisibles pour un humain dans le code sans perdre les avantages de **doxygen**.

I. Couche métier

La couche métier est la partie essentielle de l'application. C'est ici que toute l'implémentation du système est réalisée ainsi que tous les algorithmes permettant l'optimisation des trajets. De plus, comme cette couche est centrale, on y trouve également les outils permettant de communiquer avec les couches inférieures et supérieures. Par conséquent, l'API base de données qui nous permet de récupérer les informations de la BDD OPTRAJ ainsi que le framework permettant de communiquer avec la couche présentation se trouvent dans cette couche.

I.1 Classes système

Les classes système regroupent tous les objets nécessaires pour la mise en place d' OPTRAJ. Parmi ces classes se trouve par exemple la classe *Worker* qui comprend toutes les informations nécessaires à la création d'un ouvrier, mais aussi, la classe *Site* qui regroupe les informations d'un chantier, la classe *Position*, *Car*, *Assignment*, *Shuttle*, *Craft*, *Qualification*, *Phase*… Toutes ces classes sont de simples classes python disposant d'accesseurs et de mutateurs permettant l'accès et la modification des objets.

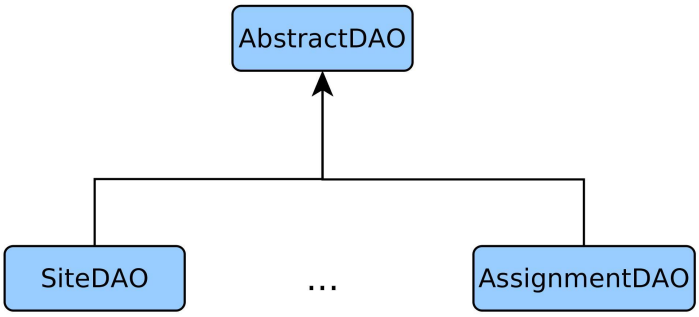
De même, toutes les classes qui sont amenées à être transmises à la couche présentation doivent être sérialisables. En effet, le format **JSON** est utilisé pour transmettre les objets python à la couche présentation. Ce format, plus léger que le format **XML**, a été choisi pour sa simplicité de mise en place. Le principe de ce format est de sérialiser un objet python grâce à la fonction *serial()*. Celle-ci transforme l'objet python en dictionnaire composé de tous les attributs de l'objet puis envoie ce nouvel objet à la couche présentation. Côté présentation, on reçoit donc un string représentant ce dictionnaire que l'on va par la suite convertir en objet du langage cible. Par exemple, pour nous, le langage cible est **JavaScript**. La fonction permettant la conversion vers **JavaScript** est *parse()*. C'est une fonction de base de **JavaScript** présente dans la plupart des langages de programmation.

I.2 API Base de données

L'interface de la base de données est réalisée en suivant le *design pattern* **DAO** (*Data Access Object*). Ainsi, l'interface BDD (package *interfacebdd*) est constituée d'une classe *AbstractDAO.py* dont héritent toutes les autres classes de l'interface. Ceci nous permet de définir des fonctions génériques pour interagir avec la BDD. Les classes filles permettent chacune de générer un objet système qui a une représentation en base de données. Ainsi, pour générer un objet *Site* (chantier), on utilisera la classe *SiteDAO*, du package *interfacebdd*.

La classe *AbstractDAO* définit les fonctions d'accès à la base de données (*getById*, *insert*, etc). Pour communiquer avec la base de données (en **MySQL**[1]), nous nous servons d'une librairie python (**PyMySQL**[2]). Cette librairie permet d'exécuter des requêtes **MySQL** et de récupérer le résultat dans un dictionnaire python. La méthode privée *_buildObject* de l'*AbstractDAO* permet de construire un objet système à partir de ce dictionnaire, en utilisant l'autre méthode privée *_createElement*. La méthode *_buildObject* est générique, et s'appuie sur la méthode *_createElement* qui est redéfinie dans chaque classe fille.

Cette classe dispose aussi de plusieurs *templates*, correspondants à des requêtes **MySQL**, que l'on complète dans les différentes fonctions de l'*AbstractDAO* (en formant les clauses par exemple). Ainsi, la classe *AbstractDAO* est entièrement générique et se sert d'informations définies dans ses classes filles.



Chacune de ses classes filles définissent deux dictionnaires python : *MAPPING_COLS* et *MAPPING_FIL*. Le premier dictionnaire fait le lien entre les colonnes des tables **MySQL** (clés du dictionnaire) et les attributs des objets du système (valeur du dictionnaire). Il est utilisé pour charger/enregistrer les données de la BDD dans un objet système. Le deuxième dictionnaire, *MAPPING_FIL*, fait le lien inverse (clés: attribut des objets système, valeurs : colonne des table **MySQL**). Ce dictionnaire est utilisé pour former les clauses *where* des requêtes **MySQL**.

Les jointures entre les tables sont aussi nécessaires à la communication entre la BDD et la couche métier. On utilise la classe *NaturalJoin.py* pour les représenter en python. Cette classe définit une jointure par l'intermédiaire de plusieurs attributs (*joinType*, *tableName*, etc) et est utilisée par certaines classes filles pour définir leurs jointures. Ceci est fait grâce à l'attribut *joins* (hérité de l'*AbstractDAO*), qui est une liste de *NaturalJoin*. On utilise ensuite cette liste pour compléter les requêtes dans les fonctions de l'*AbstractDAO*.

Enfin, certaines classes filles ont besoin de redéfinir des fonctions de l'*AbstractDAO*, notamment *SiteDAO*. Nous avons limité ces redéfinitions, qui nuisent à la généricité du design pattern **DAO**.

L'API offre la possibilité de ne charger que partiellement des données de la BDD. En effet, afin d'améliorer les performances de la base de donnée, nous avons implémenté un *lazy loading* permettant de ne charger que les jointures demandées. Ainsi, deux paramètres sont passés aux fonctions de chargement de l'API : le booléen *lazy*, et la liste de jointures *partLazy*. Si le premier de ces paramètres est à *true*, alors l'API ne chargera que les jointures indiquées dans la liste *partLazy*. Ce *lazy loading* nous a, par exemple, été très utile lors des requêtes chargeant de nombreux chantiers. En effet, nous avons diminué d'un facteur cinquante (ou plus) le temps d'exécution des requêtes concernant le chargement de tous les chantiers de la base de données.

[1] https://www.mysql.fr/
[2] https://pypi.python.org/pypi/PyMySQL/0.6.1

Il est aussi important de noter que la connexion à la base de données est gérée dans la classe *Connexion.py*. Cette classe offre des fonctions permettant de se connecter et de se déconnecter à notre BDD, en utilisant les fonctions de connexions de la librairie **PyMySQL.** Ainsi, pour utiliser les fonctions de l'*interfacebdd*, il est important d'ouvrir une connexion à la BDD dans un bloc *try/catch*, puis d'appeler les méthodes voulues dans ce bloc.

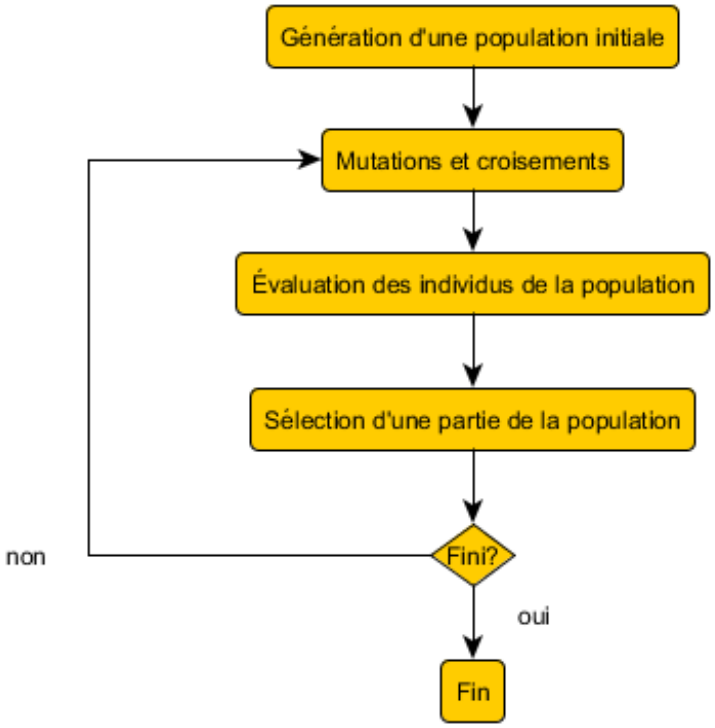
I.3 Framework Flask

La communication entre la couche métier et la couche présentation se fait grâce au framework **Flask**[3] (version 0.10.1). Ce framework permet d'associer une fonction à une url. Grâce à cela, il nous suffit d'envoyer une requête sur une url (par exemple */site/all/* pour recevoir la liste de tous les chantiers présent dans la base de données). Ce framework est vraiment très pratique et on peut très facilement rajouter des url afin d'avoir des fonctionnalités supplémentaires côté interface. Pour exemple, voir l'annexe n°2.

I.4 Optimisation

L'optimisation des trajets se fait en python en utilisant le module **DEAP**[4]. Il s'agit d'une bibliothèque implémentant divers algorithmes génétiques en se basant sur de la redéfinition de fonction génériques (mutation, sélection,…)

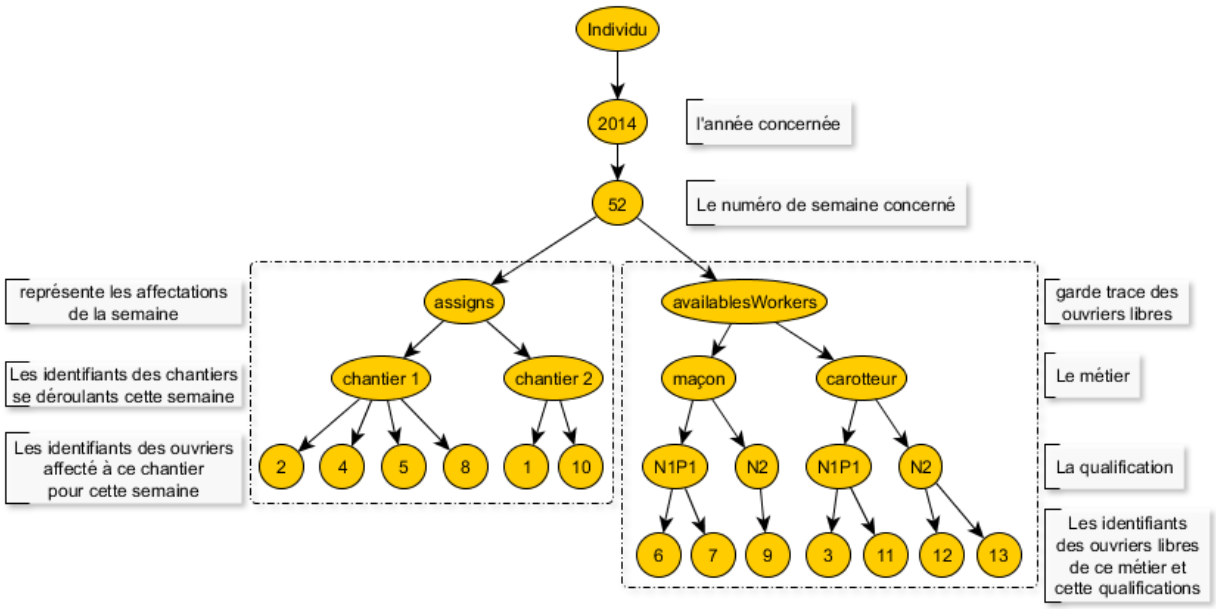
On utilise un algorithme génétique dont le principe général est rappelé ci dessous.



[3] http://flask.pocoo.org/
[4] http://code.google.com/p/deap/

Forme des individus

Les individus sont des dictionnaires python représentant des arbres. Leur forme globale suit cette organisation :



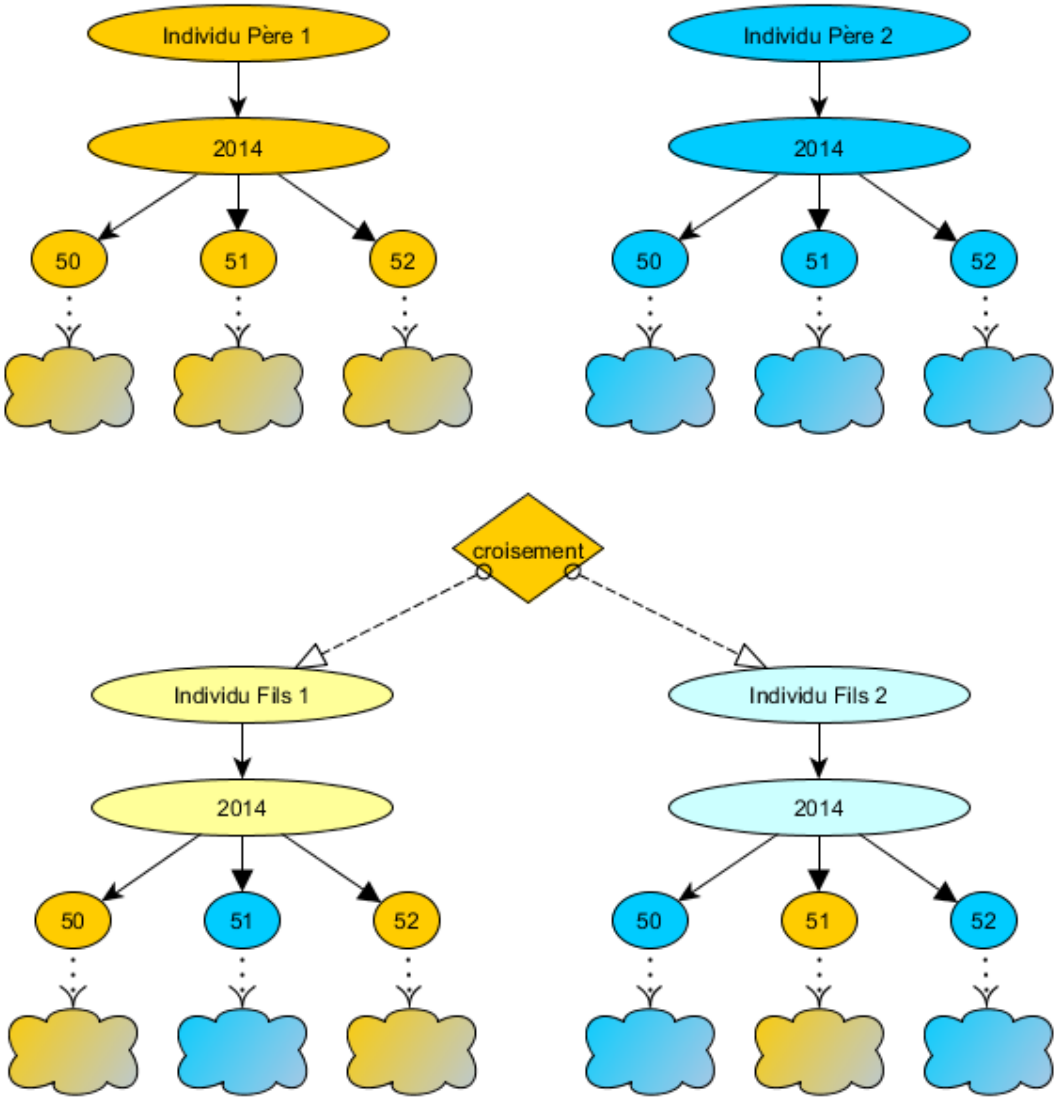
Toutes les clés sont soit des entiers (*année*, *numéro de semaine*, *numéro de chantier*, *numéro d'ouvrier*) soit des chaînes de caractères (*assigns* et *availablesWorkers*). Le sous arbre *availablesWorkers* garde la trace de l'ensemble des ouvriers non affectés et donc libres pour la semaine concernée. Ils sont classés par métier (*craft*) et qualification (*qualif*). Les ouvriers affectés à un chantier donné ne sont pas classés et sont rangés ensemble dans un set dans le sous arbre *assigns*. On utilise une structure très similaire, le *must* afin de savoir quelles sont les affectations devant impérativement être présente. Ce sont celles déjà validées en base et qui ne doivent pas être modifiées par l'algorithme. Cette structure est initialisée lors de l'appel à l'algorithme et est utilisée pour initialiser les individus lors de leur création ou restreindre leurs possibilités de mutations. Un individu représente un ensemble d'affectations valides, c'est à dire que les besoins des chantiers sont remplis autant que possible et qu'aucun ouvrier n'est affecté à deux endroits au même moment. La pertinence des affectations n'entre pas en compte pour déterminer la validité d'un individu.

Croisement

Le croisement de deux individus est relativement simple. Deux semaines distinctes sont complètement indépendantes l'une de l'autre. On peut donc en modifier une sans que cela influe sur la validité de l'individu.

Le croisement est un croisement multi points : deux nouveaux individus (les fils) sont créés. Pour chaque semaine (un gène), le fils 1 reçoit aléatoirement le sous arbre correspondant soit chez le père 1, soit chez le père 2 (le fils 2 reçoit alors le sous arbre de l'autre parent).

Par exemple, l'image ci dessous représente un croisement où le fils 1 reçoit les sous arbres du parent 1 pour les semaines 50 et 52, et le sous arbre du parent 2 pour la semaine 51 (inversement pour le fils 2).

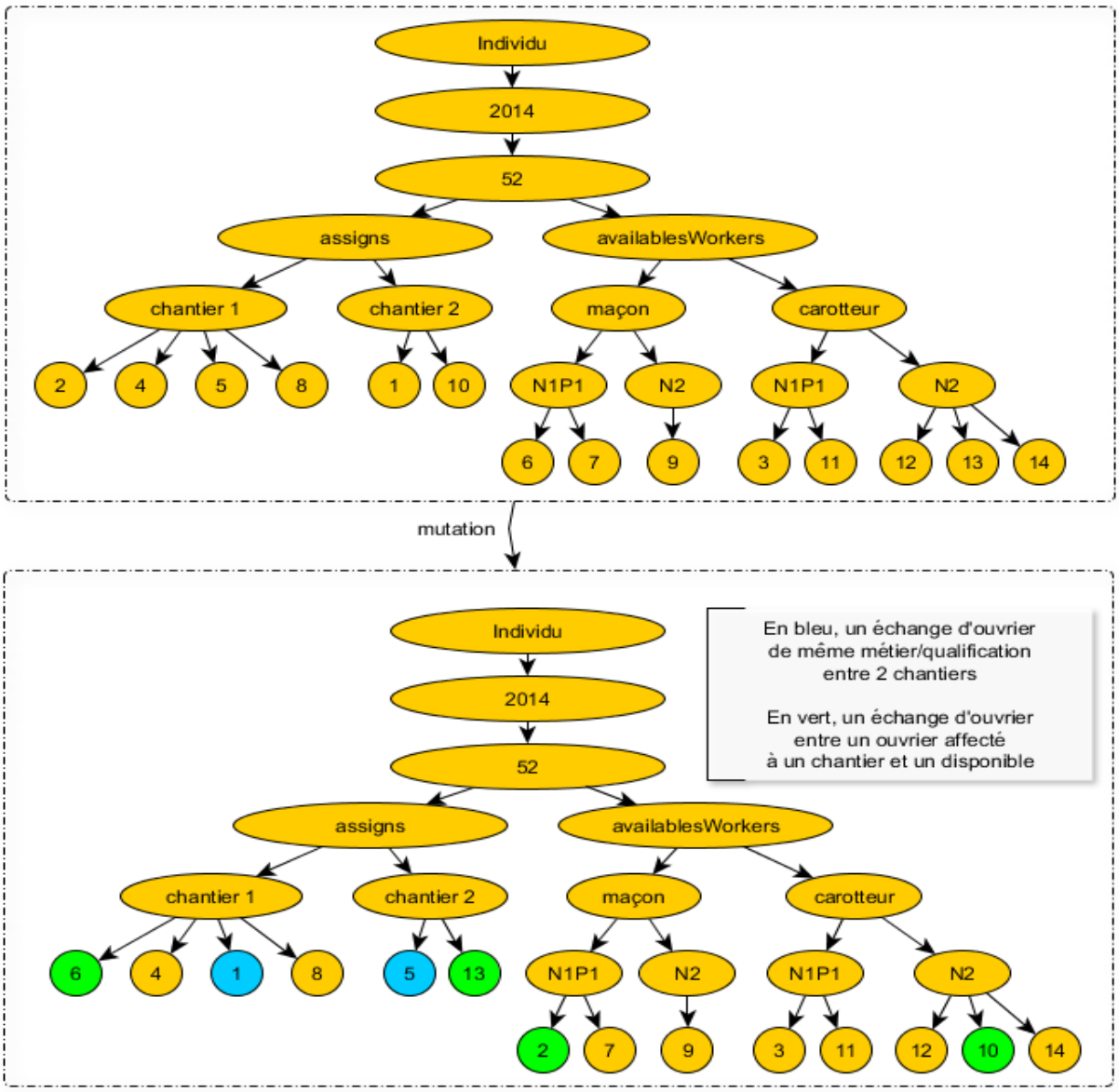


Mutation

Pour la mutation, on cherche à changer les affectations de chaque semaine. Ce traitement se fait en deux étapes:

- premièrement, on retire aléatoirement des ouvriers affectés aux chantiers. Ces ouvriers sont réinjectés au fur et a mesure dans *availablesWorkers*. La seule contrainte à ce niveau est qu'on ne peut pas retirer des ouvriers étant dans le *must*.
- deuxièmement, on repasse sur chaque chantier afin de remplir à nouveau ses affectations pour satisfaire ses besoins.

De cette manière, on rend possible l'échange d'ouvrier d'un chantier à un autre, aussi bien que l'échange avec un ouvrier non sélectionné au départ.



Évaluation

Trois critères d'évaluations sont utilisés : la distance du conducteur de la navette au chantier, la distance cumulée pour chaque navette de trajet des ouvriers vers le trajet de la navette, et le nombre de changement d'affectations pour chaque ouvrier.

Les distances calculées sont des distances à vol d'oiseau, uniquement des tracés en ligne droite.

A partir de l'ensemble des affectations pour un chantier, on sélectionne l'ouvrier le plus éloigné et on le désigne comme conducteur. On récupère ensuite tous les ouvriers susceptibles d'être sur le trajet en définissant une distance maximale de ramassage. La distance de trajet de navette est multipliée par le

nombre d'ouvriers affectés à la navette divisé par cinq, pour prendre en compte le fait que, réellement, il faudra plusieurs véhicules.

Le nombre de changements d'affectation quant à lui se calcule simplement en incrémentant une valeur à chaque fois qu'un ouvrier est sur un chantier X durant la semaine N mais sur un chantier Y à la semaine N+1

Ces trois valeurs sont ensuite normalisées pour être comparables, puis multipliées par leurs coefficients afin de n'avoir au final qu'une seule valeur, le score de l'individu.

Sélection

La sélection se fait en se basant sur le score. À chaque itération, on conserve X individus qui serviront de base pour mutation et croisement lors de la prochaine itération.

Sur ces X individus, la moitié est composée des meilleurs individus de la population actuelle, le reste est tiré au hasard parmi les individus restants. Cela permet de ralentir la convergence du système, de garder de la diversité dans nos individus afin de tirer mieux parti du concept d'évolutions.

I.5 Importation des données

La base de données de l'application n'est pas celle utilisée de Cardinal, régulièrement mise à jour. Afin de rendre l'application réellement utilisable nous avons décidé de créer un système d'importation des ouvriers depuis la base de données de Paie de Cardinal. Seuls les ouvriers sont mis à jour car les autres données n'étaient pas en base. Régulièrement, l'autre base de données se charge de déposer un fichier sur la machine hébergeant l'application. Ensuite le script *GenerateWorkers.py* est exécuté périodiquement pour mettre à jour la BDD.

Format et utilisation des données

Deux fichiers doivent être à l'emplacement */vagrant/DonneesBDD/* de la machine. Ils doivent être encodés en UTF-8 sans BOM.

Le premier est *reglesMetiers.txt*. Il définit les correspondances entre les métiers côté base de données de paie, et ceux définis dans le logiciel. Chaque ligne en divisée en deux par deux points, à gauche de celui-ci on a un ou plusieurs métiers de la base de paie et à droite le métier que l'on veut faire correspondre. Ces métiers sont entre guillemets séparés par un espace. Par exemple si on veut faire correspondre le métier "Développeur" et le métier "Informaticien stagiaire" au métier "Informaticien" on aura :

"Développeur" "Informaticien stagiaire": "Informaticien"

Le deuxième est *salaries.csv* ce fichier contient les ouvriers amenés à travailler sur les chantiers. C'est ce fichier qui sera régulièrement mis à jour. Il est défini simplement, chaque ligne représente un ouvrier, ses informations sont séparées par des virgules. Elles sont dans l'ordre : nom, prénom, date de naissance (au format **SQL**), rue, code postal, ville métier, qualification.

Fonctionnement

On utilise deux fichiers python. Le premier *checkFile.py* nous permet de vérifier si les deux fichiers existent et sont bien construits. L'objet *checkFile* a donc comme attributs le chemin du fichier et le type (CSV ou Rules).

Ensuite on a le fichier principal *GenerateWorkers.py.* Il permet de créer ou de modifier les ouvriers par rapports à ceux extraits dans la base. Sa méthode *main* vérifie d'abord que les fichiers sont là et bien construits. Ensuite on crée un métier *non référencé*, métier sera en lien avec tous ceux non trouvés dans les règles. Pour chaque ligne de mon fichier nous allons regarder si on a un ouvrier du même nom, prénom et même date de naissance, seules informations étant fiables dans cette situation.

Si ce n'est pas le cas, on crée un ouvrier. Pour l'adresse, on va chercher via la librairie *pygeocoder*, qui permet d'utiliser les services de **Google Maps** facilement, les coordonnées de l'adresse complète de l'ouvrier, si il n'y a pas de résultat on prend celles de sa ville. À noter qu'on attend une seconde après la requête pour éviter que **Google Maps** nous bannisse de ses services temporairement. Pour le métier et la qualification on va retourner l'identifiant de la base de données. Ensuite on insère l'ouvrier. Si c'est le cas, on modifie l'ouvrier, on le récupère en changeant obligatoirement son métier, sa qualification et sa position.

II. Couche présentation

II.1 Importation des données

Voir schéma de la BDD en annexe 1 pour voir le détail de ce qu'est un *Consumer*.

Niveau d'administration

1. lecture et écriture : L'utilisateur a accès à toutes les pages sauf l'onglet administrateur, il peut donc gérer chantiers, ouvriers, navettes, optimisation…
2. lecture seule : L'utilisateur à seulement les droits de consultation, il ne peut donc rien modifier.
3. admin : Même droit que lecture et écriture avec la gestion des utilisateurs.

Cryptage MD5

Comme tout système d'identification, le mot de passe ne doit pas être reconnaissable une fois stocké dans la base de données, nous avons donc crypté le mot de passe avec le cryptage **MD5** qui crypte n'importe quelle chaîne de 32 caractères en une chaîne unique de caractères.

Session

Un utilisateur peut rester connecté à l'aide du système de session de **JavaScript** (*sessionStorage*), ce système permet de créer des variables de session qui sont remises à zéro quand le navigateur est fermé, elles peuvent tout de même être modifiées en cas de déconnexion ou de changement d'utilisateur. Pour notre système nous utilisons cinq variables:

- *alreadyco* : Permet de savoir si un utilisateur s'est connecté au moins une fois
- *isco* : Permet de savoir si un utilisateur est actuellement connecté.
- *lvladminreq* : Permet de stocker le niveau de l'utilisateur connecté.
- *logname* : Permet de stocker le nom de l'utilisateur connecté.
- *logfirstname* : Permet de stocker le prénom de l'utilisateur connecté.

Couche métier

Pour le système d'identification nous avons rajouter la requête **Flask** *connectConsumer()* qui, avec le login et mot de passe de l'utilisateur, permet de se connecter.

Une fois connecté, la requête *nameConsumer()* stock dans les variables de session précédemment citée, le nom et le Prénom de l'utilisateur pour ainsi l'afficher.

Couche interface

La page contenant le formulaire d'identification est *connexion.php,* une fois connecté les variables de sessions sont mis à jour, ce formulaire n'est plus accessible après.

La fonction *deco()*, permettant de réinitialiser les variables de session, peut être appelé dans n'importe quelles page de l'application.

Le fichier *session.js* est appelé depuis *index.php*, et permet de cacher ou d'afficher les différents onglets de l'application selon la connexion et le niveau d'administration.

Les fichiers *logadmin.js*, *logadminreq.js*, et *logadminup.js*, teste respectivement si un utilisateur est connecter, qu'il ne soit pas de niveau lecture seul et qu'il soit de niveau admin, dépendant des droits requis pour la page appelé.

Le fichier *removeButton.js* permet de cacher les boutons ''ajouter un .....'' si l'utilisateur connecté est de niveau lecture seule.

II.2 Système d'administration

Niveau requis

Seul un utilisateur de niveau admin a accès à la gestion des utilisateurs. L'onglet administrateur contient différentes pages qui permettent la gestion des utilisateurs.

Requête

Les requêtes *allConsumer()*, *createAdmin(), deleteConsumer()* et *updateConsumer()* permettent la gestion des utilisateurs.

Fichiers

Seul des fichiers de la couche interface sont ajoutés pour le système d'administration, dont voici la liste et l'arborescence des fichiers utilisés pour le système d'administration.

consultAdmin.php : Contient la liste des utilisateurs, généré dynamiquement avec la requête *allConsumer()* depuis *consultAdmin.js*, ainsi qu'un bouton pour créer un utilisateur se dirigeant vers la page *createAdmin.php*.

createAdmin.php : Contient le formulaire de création d'un utilisateur, que l'on valide avec le bouton créer appelant la requête *createAdmin()* depuis *createAdmin.js*, un formulaire ne peut être validé si un champ est vide ou si le login entré existe déjà dans la base de données.

editAdmin.php : Affiche les informations de l'utilisateur sélectionné, cet utilisateur est stocké dans une variable locale **JavaScript**, on fait apparaître le formulaire de modification si on clique sur une des informations en positionnant le curseur sur cette information.

On peut soit confirmer le formulaire avec le bouton Valider les changements appelant la requête *updateConsumer()*, soit supprimer cet utilisateur avec le bouton “Supprimer” l'utilisateur appelant la requete *deleteConsumer()*, soit être dirigé vers la page *editPwd.php* pour modifier le mot de passe avec le bouton Modification du mot de passe, ici aussi un formulaire ne peut être validé si un champ est vide ou si le login entré existe déjà dans la base de données.

editPwd.php : L'utilisateur est toujours dans une variable locale, on peut ainsi comparer le mot de passe avec le mot de passe crypté fourni par l'utilisateur dans le champ ancien mot de passe ainsi le mot de passe peut être modifié si le nouveau est différent de l'ancien avec la requête *updateConsumer()*.

II.3 Proxy de communication

La nécessité d’utiliser un proxy est due au fait que la partie **JavaScript** et la partie **Flask** ne sont pas situés au même endroit. En effet, le **JavaScript** est exécuté côté client alors que le **Flask** l’est côté serveur. Il nous faut donc un mécanisme permettant d’échanger les données entre ces deux parties. La solution est d’utiliser l’architecture **AJAX** pour dialoguer de manière asynchrone avec le serveur. Cependant, **Flask** ne supporte pas les requêtes provenant d’une url différente de la sienne (requête cross-domain). Ainsi, **AJAX** n’est pas suffisant et il nous faut un autre outil permettant d’envoyer une requête depuis le même domaine que **Flask**. Pour cela, on utilise un proxy.

Ce proxy est décomposé en deux parties. La première est la partie **JavaScript** qui envoie la requête vers le proxy à l’aide d’**AJAX**. La deuxième partie est le proxy en lui même, qui récupère la requête provenant de **JavaScript** et qui la transfère vers **Flask**.

Partie JavaScript

La partie JavaScript du proxy s’appuie sur **AJAX** pour permettre d’envoyer une requête de manière asynchrone au serveur **Flask**. **AJAX** a la particularité d’être asynchrone, ce qui nous permet d’envoyer un requête vers un serveur et de ne pas bloquer le client en attendant la réponse. Ce n’est que lorsque la réponse arrivera que le client traitera les données à l’aide d’une fonction. Nous avons implémenté cela grâce à une fonction *request(url, func, data)* qui envoie une requête vers *url* avec *data* en paramètre si celui-ci n’est pas vide et qui appelle la fonction *func* lorsque le serveur rend une réponse. Cette fonction s’occupe elle-même de créer une requête de type GET si *data* est vide, et une requête de type POST si *data* n’est pas vide.

Partie PHP

La partie **PHP** est le coeur même du proxy. Grâce à **PHP**, nous pouvons lancer des requêtes depuis le même domaine que **Flask**, qui peut donc reçevoir les requêtes et les traiter. Ce proxy se contente de

récupérer l'url de destination, c'est à dire une url vers le serveur **Flask**, ainsi que les possibles données à transmettre dans le cas d'une requête POST. Le proxy s'appuie sur *curl* pour former les requêtes POST et sur la fonction *file_get_contents* pour le cas simple des requêtes GET.

II.4 Création, consultation et modification de données

La partie consultation et modification de données est relativement similaire pour les trois différentes catégories que sont Chantier, Ouvrier et Navettes. En effet, dans les trois parties les traitements restent similaires malgré quelques spécificités expliquées à la fin de cette section.

Création

Les pages de création sont regroupées dans les fichiers ayant le nom *create* suivi du nom de l'objet concerné (ex: *createSite.js*, *createWorker.js*, *createShuttle.js*). Dans ces fichiers, on y trouve principalement des fonctions qui affichent les différents champs correspondant aux attributs de l'objet concerné. Ces champs sont regroupés dans un formulaire qui est envoyé via une requête **Flask** vers la base de donnée pour construire l'objet. On y trouve également les fonctions de vérification des champs pour ne pas permettre l'envoi du formulaire si un des champs n'est pas rempli.

Consultation

Pour la partie consultation, on s'occupe d'abord de récupérer une liste de ce qui nous intéresse (les chantiers, les ouvriers ou les navettes) puis on les affiche dans un tableau trié par ordre (alphabétique pour les ouvriers et les navettes, et par numéro de chantier pour les chantiers). Ce tableau est agrémenté d'un champ de recherche permettant de filtrer les objets affichés.

Ce champs de recherche s'appuie sur une fonction *filter(e, tabElement)* qui va construire la liste des objets à afficher selon le texte tapé dans le champs recherche. Ce champ de recherche permet de faire une recherche multi-critère. Ainsi, si l'on souhaite par exemple afficher tous les ouvriers maçon de qualification N1P2, il nous suffit de taper "maçon n1p2" dans le champ.

Enfin, lorsque l'on clique sur une des lignes, nous sommes redirigé vers la page d'édition de l'objet concerné. Cette transition est possible grâce aux variables de session **JavaScript** dans lesquels nous mettons l'id de l'objet concerné.

Toutes ces fonctions sont regroupées dans les fichiers commençant par *consult* suivi de l'objet concerné (ex: *consultSite.js*, *consultWorker.js*, *consultShuttle.js*).

Modification

La page de consultation se contente de récupérer l'objet concerné grâce à l'id de celui-ci qui est passé dans la variable de session **JavaScript**. On fait donc une requête vers le serveur **Flask** qui récupère l'objet. Ensuite, on présente les différents attributs de chaque objet en permettant leur modification dès que l'on clique sur un des champs.

C'est la fonction *modifInput* qui nous permet de transformer les informations de l'objet en formulaire et d'afficher les boutons permettant de valider les modifications effectuées.

On trouve également les fonctions qui permettent de gérer les adresses des objets afin de les afficher sur les **Google Maps**. On peut par exemple citer les fonctions *checkAddress* qui vérifie si l'adresse de l'objet est une adresse valide et *addressFail* qui nous affiche un message lorsque l'adresse n'est pas valide.

Ces différentes fonctions sont regroupés dans les fichiers **JavaScript** de nom *edit* suivi de l'objet concerné (ex: *editSite.js*, *editWorker.js*, *editShuttle.js*).

Différences selon le type d'objet

Chantiers

Il y a deux fonctionnalités spécifiques à la page de modification d'un chantier. En effet, dans cette page, la **Google Map** permet d'avoir un affichage plus précis des navettes attribuées à un chantier pour une semaine donnée.

Pour cela, la fonction *expandMap* est utilisée afin d'afficher une carte qui prend toute la place à l'écran. Celle-ci ne s'occupe que de cacher la div du corps de la page, d'afficher celle qui contiendra la grande carte (*BigMap*). Elle appelle ensuite la fonction *loadBigMap* qui va pouvoir charger la BigMap dans la div et y afficher le chantier sur cette carte.

Ensuite, on récupère l'ensemble des ouvriers et les différentes navettes de la semaine courante grâce à ces deux requêtes :

```
templates/proxy.php?url=http://localhost:5000/worker/assigned/bysite/byweek/
templates/proxy.php?url=http://localhost:5000/shuttle/bysite/byweek/
```

L'affichage se fait grâce à *showWorkersOnMap* qui appel d'abord la fonction *createDirectionsRen* : celle-ci va paramétrer les itinéraires pour chaque navette et les créer avec la fonction *route*. Enfin on affiche les cercles représentant les ouvriers (*addCircle*) et les lignes (*addLine*) reliant les ouvriers à leur point de rendez-vous.

La deuxième fonctionnalité spécifique aux chantiers est la présence d'un tableau représentant les besoins du chantier concerné. Ceux-ci sont classés par numéro de semaine sur une durée de quatre mois (dix-

sept semaines) et affichés par métier puis par qualification. Les fonctions nécessaire à la création de ce tableaux sont regroupé dans le fichier *needOfSite.js*.

Pour construire ce tableau, on commence par faire le header grâce à la fonction *initHeader* qui prend en paramètre la date de début du tableau et le nombre de mois à afficher. Elle crée un header composé d'une première ligne correspondant à l'année, puis une seconde qui regroupe les mois et enfin une ligne composée des numéros de semaines correspondantes.

On construit ensuite les différentes lignes qui correspondent aux différents métiers existant en base et récupérées comme les qualifications grâce à une requête vers le serveur **Flask**.

On peut ensuite continuer en affichant les différents besoins déjà présents en base de données. Pour cela, on parcourt toutes les phases associées au chantier courant puis, pour chaque phase, tous les besoins qui sont définis. Pour faciliter les traitements, on a décidé de créer un dictionnaire contenant tous les besoins triés par numéro d'année, numéro de semaine, métier et qualification afin de retrouver plus facilement l'information par la suite. De même, chaque case du tableau se voit attribuer le numéro de semaine correspondante (de 0 à 16) toujours pour faciliter les traitements. Enfin, on crée les différentes lignes correspondant aux différentes qualification présentes en base pour chaque ligne métier. Il ne reste plus qu'à parcourir les différentes cases et à y mettre les besoins associés.

Enfin, plusieurs fonctions permettent de rendre ce tableau plus agréable. Par exemple, la fonction *computeCraft* permet de calculer la somme des besoins déjà définis pour chaque métier. Ainsi, dès que l'utilisateur rentre une valeur dans une case, celle-ci provoque la mise a jour de la somme sur la ligne métier correspondante. De même, on fait la même opération pour la somme totale des besoins définis sur une semaine. Celle-ci est affichée dans la ligne besoin en haut du tableau.

Il existe également une fonction *copyValue* qui permet de recopier une valeur rentrée par l'utilisateur d'une case à l'autre grâce à l'appuie sur la touche tabulation.

Navettes

La spécificité des navettes vient du fait que l'on peut afficher toutes les navettes ou alors seul les navettes affecté à un chantier pour une semaine donnée. Pour la première liste, le traitement est le même que pour les autres objets. Cependant, pour la seconde, il nous faut récupérer les navettes affectées à un chantiers selon une semaine. Nous utilisons pour cela une requête vers **Flask** qui nous retourne seulement les navettes concernées. De même, on doit pouvoir changer de semaine pour pouvoir voir quelles navettes sont affectées pour les semaines suivantes. Pour cela, nous utilisons une fonction *changeNumWeek* qui met à jour les navettes à afficher selon la nouvelle semaine définie.

II.5 Timeline

La timeline nous permet de présenter les différents chantiers actifs à ce jour. Cette partie est comme la plupart des autres codée en **JavaScript** et se trouve dans le fichier *scriptTimeline.js*. Plusieurs fonctions sont utilisées pour permettre d'avoir une timeline fonctionnelle et agréable. Le principe est de récupérer tout les

chantiers en base et de les classer dans deux tableaux **JavaScript** : *chantiersVisibles* et *chantiersInvisibles*. Le premier comme sont noms l'indique contient tous les chantiers qui seront visible sur la timeline, c'est à dire ceux qui ont une date de début de chantier inférieure à la date de fin de timeline et une date de fin supérieure à la date de début de timeline. Au contraire, le second tableau reprend tout les chantiers qu'on ne verra pas dans la timeline. Toutes ces opérations sont effectuées dans la fonction *refreshChantier*.

On garde tous ces chantiers pour pouvoir par la suite naviguer dans la timeline. En effet, si l'utilisateur souhaite aller voir dans le passé ou dans le futur, on doit pouvoir afficher les anciens et futur chantiers. Pour cela, deux fonctions *changeBeginTime* et *changeEndTime*, changent respectivement le début et la fin de la timeline. Ainsi, il ne reste plus qu'à mapper les boutons de navigation sur ces fonctions pour pouvoir avoir une navigation complète dans la timeline. Nous avons opté pour une navigation qui décale la timeline d'un cran vers la droite ou vers la gauche. Il faut donc faire appel à *changeBeginTime* et *changeEndTime* à la suite d'un appuie sur une des touches de navigation.

De même, nous pouvons également modifier le nombre de mois affiché par la timeline. Par défaut, nous affichons douze mois mais cette valeur peut être changée pour n'afficher que quelque mois ou au contraire plusieurs années. Cette opération fait appel à la fonction *changeEndTime* qui nous permet de modifier la fin de timeline et donc le nombre de mois affiché.

II.6 Optimisation

La page d'optimisation sert à améliorer les affectations des ouvriers sur les chantiers en fonction de leurs distance par rapport à ces derniers. Ceci s'applique en lançant l'algorithme d'optimisation qui se chargera de proposer des affectations optimales. Il est évident qu'il est possible d'effectuer ses propres affectations en ne prenant pas en compte les propositions de l'algorithme.

Cette page d'optimisation est essentiellement composée de code **JavaScript** (**JavaScript** pur et fonctions **JQuery**) afin d'obtenir le plus de dynamisme possible. Nous allons voir ce qui se passe au chargement de la page, à l'appui du bouton "lancer les calculs", "valider les choix", "exporter en .xls" et enfin comment modifier/créer des affectations manuellement. Le code de manipulation est essentiellement présent dans le fichier *showAffect.js* et *afterLoad.js*.

Sur lancement de la page, deux requêtes permettant d'obtenir l'ensemble des chantiers et l'ensemble des ouvriers sont lancés sur le serveur :

```
templates/proxy.php?url=http://localhost:5000/site/all/lazy/ GET
templates/proxy.php?url=http://localhost:5000/worker/all/ GET
```

Le résultat de la première requête (en **JSON**) est parsé pour ensuite être intégré dans le tableau des affectations (le nom de l'ouvrier, sa qualification ainsi que sa ville de résidence).

Le résultat de la deuxième requête est utilisé pour garder en mémoire les informations de tous les chantiers qui seront utilisées lorsque l'utilisateur souhaitera consulter une affectation d'un ouvrier (explication plus tard dans le document).

Vient ensuite une troisième requête qui vise à récupérer l'ensemble des affectations à partir de la semaine actuelle de l'année en cours jusqu'à la dernière semaine (on effectue une optimisation sur 17 semaines).

templates/proxy.php?url=http://localhost:5000/assignments/part/ POST

Le résultat de cette requête est récupéré pour être intégré dans le tableau des affectations. Chaque affectation de chaque ouvrier en fonction de la semaine est intégrée, via JavaScript (fonction *fillTableWithExistingAssignments*), dans la case correspondante.

Pour permettre d'afficher une popup sur clique d'une case d'affectation, une fonction permettant de manipuler le DOM applique (après le tag "***GESTION DES POPUP"*** dans le fichier *showAffect.js*), sur clique, la création et l'affichage de la popup contenant les informations nécessaires en fonction du type d'action :

- création d'une affectation (Nom de l'ouvrier, numéro de semaine, chantier à choisir et la validation)
- modification d'une affectation (Nom de l'ouvrier, numéro de semaine, chantier en cours et la validation).

Les fonctions permettant de gérer les affectations à travers les popup sont :

- *showPopup(elementClicked)* : affiche une popup en fonction du clique d'un élément (type <td></td> du tableau)
- *validatePopup()* : valide la création/suppression/modification d'une affectation

Tout comme le logiciel ***Excel*** de la suite office, il est possible de "copier" les affectations, c'est à dire permettre la possibilité d'affecter un ouvrier plusieurs semaines sur un même chantier. Pour cela, l'appui sur la touche clavier "*tabulation*" permet de copier l'affectation d'origine sur les autres semaines souhaitées contiguës. Il sera possible, via la tabulation, d'écraser une affectation déjà existante (cela évitera à l'utilisateur de supprimer l'affectation existantes pour ensuite en recréer une manuellement).

La fonction nécessaire pour ces manipulations est la suivante :

- *nextTd(event)* : capture l'évènement déclencheur de la tabulation, créer un clone de l'affectation d'origine, effectue les modifications de semaine différente (semaine +1), met à jour le tableau d'affectation qui sera envoyé en base de données (sur clique *Valider les choix* de la page) puis finit en effectuant les modifications **CSS** permettant une affectation visuelle sur le tableau.

Lors du clic sur une semaine dans le header du tableau, une popup s'ouvre pour afficher les besoins d'un chantier sélectionné ainsi que son nombre d'affectations courantes. Il suffit de sélectionner un chantier dans la liste déroulante de la popup ouverte pour avoir accès à ces informations. Si aucun besoin n'est défini, alors un message est affiché dans la popup pour indiquer qu'il n'y a aucun besoin de défini pour cette semaine.

Les fonctions permettant de visualiser les besoins à travers la popup sont :

- *showSites(elementClicked)* : affiche une popup en fonction du clique d'un élément (type <td></td> du tableau)
- *showNeeds(needs)* : affiche les besoins en fonction du chantier et de la semaine passés en paramètres.

III. Base de données

Le fichier *creation.sql* permet de créer le squelette de la base de données puis d'ajouter les informations nécessaires à chaque table.

Ce fichier créé la BDD de nom *OPTRAJ_BDD*, et un utilisateur (login: *Client*, password: *password*). Après la création des tables, ce fichier les remplis avec le contenu des différents fichiers *dataXXX.csv* (où XXX est un nom de table, comme *dataSite.csv* par exemple).

Ce fichier génère donc la base de données présentée en annexes (Annexe 1).

IV. Tests Jenkins

Afin de tester notre couche métier, nous avons décidé de déployer un serveur **Jenkins**[5] sous **Vagrant**. Ainsi, le fichier *OPTRAJTest.xml* (dossier */vagrant/jenkins*) décrit le job **Jenkins** associé à notre projet. Ce fichier est chargé automatiquement dans le serveur **Jenkins** déployé sous **Vagrant** à la création de la machine virtuelle.

Le job **Jenkins** ainsi créé va effectuer tous les tests du dossier */vagrant/optraj.istic.univ-rennes1.fr/src/tests*, et fournir un rapport contenant le bilan de la batterie de tests ainsi que la couverture de code (37% à l'heure actuelle). Ce rapport est visible en sortie console depuis l'interface web du serveur **Jenkins**.

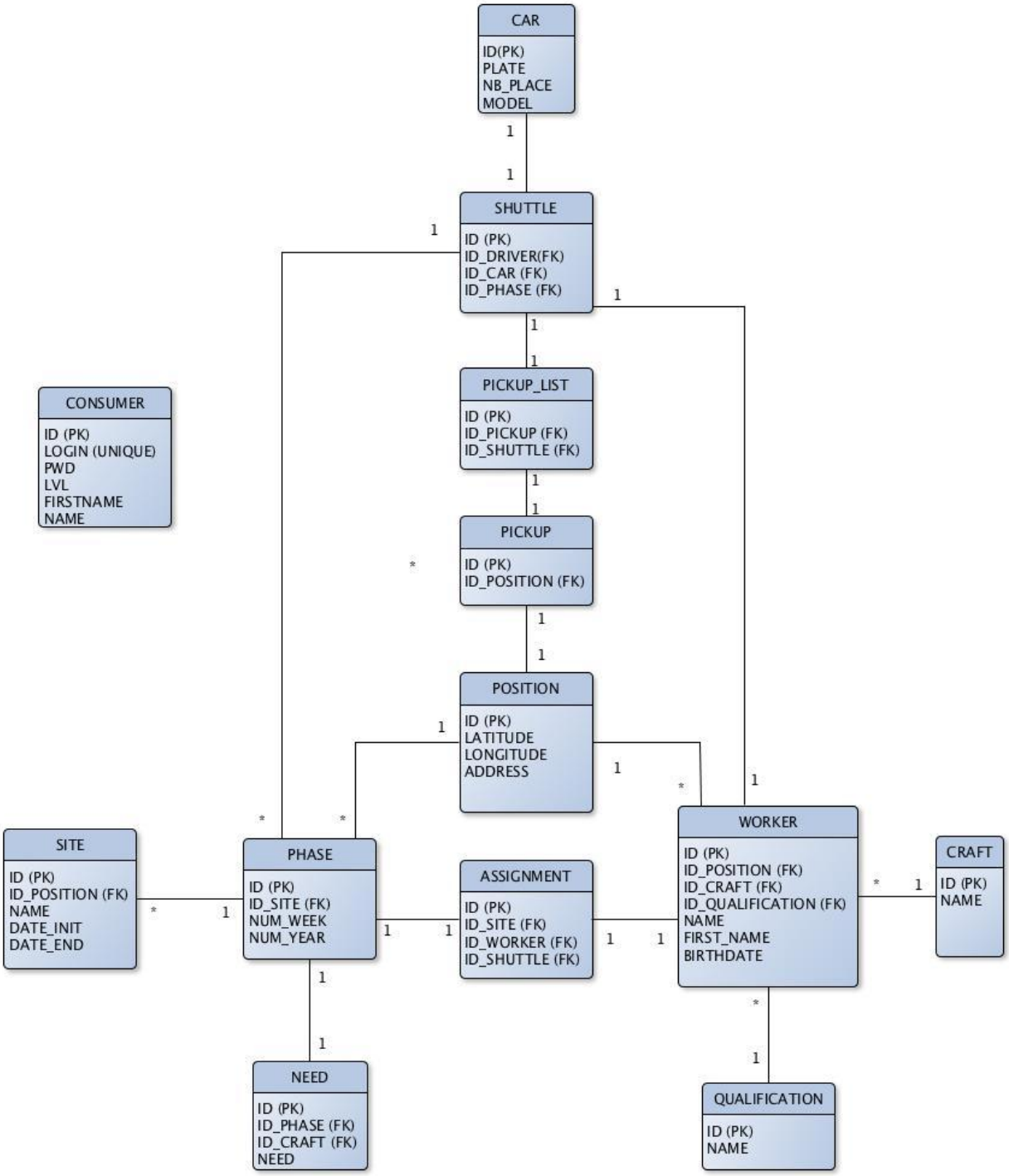
Pour le moment, nous lançons ces tests à la main depuis l'interface web de **Jenkins**, mais il est prévu d'implémenter un *Git Hook* afin de lancer ces tests automatiquement lors d'un *git pull*.

[5] http://jenkins-ci.org/

Annexes

Annexe 1 – Schéma BDD

Cf *II.1 Importation des données*.



Annexe 2 – Exemple Flask

Cf *I.2 Framework Flask*.

```
@app.route('/site/byid/', methods=["POST"])
def siteById():
    try:
        site = SiteDAO()
        conn = Connexion().connect()
        if request.method == 'POST':
        s = site.getById(conn, False, [], request.form['num']).serial()
        return json.dumps(s, encoding="utf-8")
    except pymysql.err.Error:
        Connexion().exception()
    finally:
        Connexion().disconnect(conn)
```

Annexe 3 – Vagrant

Cf *Les outils utilisés*.

Vagrant s'appuie sur **VirtualBox** pour fonctionner. Le principe de cet outil est de déployer une machine virtuelle avec tous les outils nécessaires à votre logiciel installé dessus. Ainsi, pour faire fonctionner le système il suffit de lancer la machine virtuelle.

La configuration d'une machine virtuelle **Vagrant** est relativement simple. Il suffit de spécifier toute la configuration du système dans un fichier *Vagrantfile*. Ainsi, pour notre exemple, nous utilisons une box **VirtualBox** basée sur **Ubuntu precise 64 bits**. Il suffit de préciser l'url de la box (beaucoup de box sont hébergé sur le site de **Vagrant**) pour que celui-ci télécharge la box et commence l'installation. L'avantage de ces box est qu'elles sont dépourvues de tout ce qui est inutile. Elles sont donc très légères mais on doit y rajouter tout ce dont nous avons besoin pour faire fonctionner notre projet.

Pour cela, on spécifie dans le *Vagrantfile* le chemin vers le fichier contenant tout les logiciel à installer sur la VM. Dans notre cas, c'est le fichier *bootstrap.sh*. Celui-ci est un simple fichier **shell** contenant une suite de commande **shell** à exécuter. Par exemple, voici les premières lignes de ce fichier :

```
sudo apt-get update
sudo apt-get -y install apache2
sudo apt-get -y install php5
sudo apt-get -y install php5-curl
```

```
sudo apt-get -y install curl
sudo apt-get -y install python2.7
```

Enfin, il est possible d'ajouter dans le Vagrantfile une redirection de certains ports afin de ne pas entrer en conflit avec la machine hôte. Nous avons par exemple redirigé le port 80 de la machine virtuelle sur le port 1234 de la machine hôte afin de pouvoir accéder depuis la machine hôte, au serveur apache présent dans la VM.

Une fois que la configuration de la machine virtuelle est terminée, il suffit de se rendre dans le dossier contenant le *Vagrantfile* et de taper la commande :

```
vagrant up
```

Ceci a pour effet de lancer la machine virtuelle. Au premier lancement, le téléchargement de la box et l'installation des outils peut prendre un certain temps. Une fois la machine lancé, il suffit d'exécuter le script de configuration **apache** et **MySQL** qui vont créer la base de donnée et mettre en marche le serveur **apache** ainsi que le serveur **Flask**. Ce script est dans le fichier *init.sh*. Ce script ne doit être lancé que lors de la première mise en route de la machine virtuelle. Pour les prochains lancements, le script *start.sh* suffira.

Voici une liste non exhaustive de commande **vagrant** :

```
vagrant halt        //pour arrêter la machine virtuelle
vagrant reload      //pour relancer la machine virtuelle
vagrant ssh         //pour accéder à la machine virtuelle via une connexion ssh
vagrant provision //pour recharger le fichier bootstrap.sh si l'on souhaite réinstaller de
                    nouveaux logiciel dans la machine virtuelle
vagrant destroy   //pour détruire la machine virtuelle si celle-ci est hors d'usage ou si
                    l'on souhaite repartir d'une nouvelle machine virtuelle
```

Annexe 4 – Extrait Doxygen

Renvoie
le score de l'individu dont l'évaluation est la plus haute dans pop

```
def src.optimisation.pythondeap.algo.min ( self,
                                           pop
                                         )
```

Évaluation du minimum dans une population.

Paramètres
pop la population à évaluer

Renvoie
le score de l'individu dont l'évaluation est la plus basse dans pop

```
def src.optimisation.pythondeap.algo.mutate ( self,
                                              individual
                                            )
```

Fonction de mutation d'un individu.

```
Une mutation consite à échanger aléatoirement au sein d'une semaines des affectations d'ouvrier.
On n'échange que des ouvriers de même métier et qualification.
L'échange est possible aussi bien entre 2 chantiers que entre un chantier et la liste des ouvriers non affectés.
```

Paramètres
individual l'individu qui doit être muté

Renvoie
un tuple de l'individu passé en paramètre

Extrait de la documentation générée par **doxygen**

Annexe 5 – Configuration minimum

Cf *Annexe 3*.

L'application OPTRAJ doit être installée sur un serveur. Sa configuration doit être au minimum :

Coté matériel

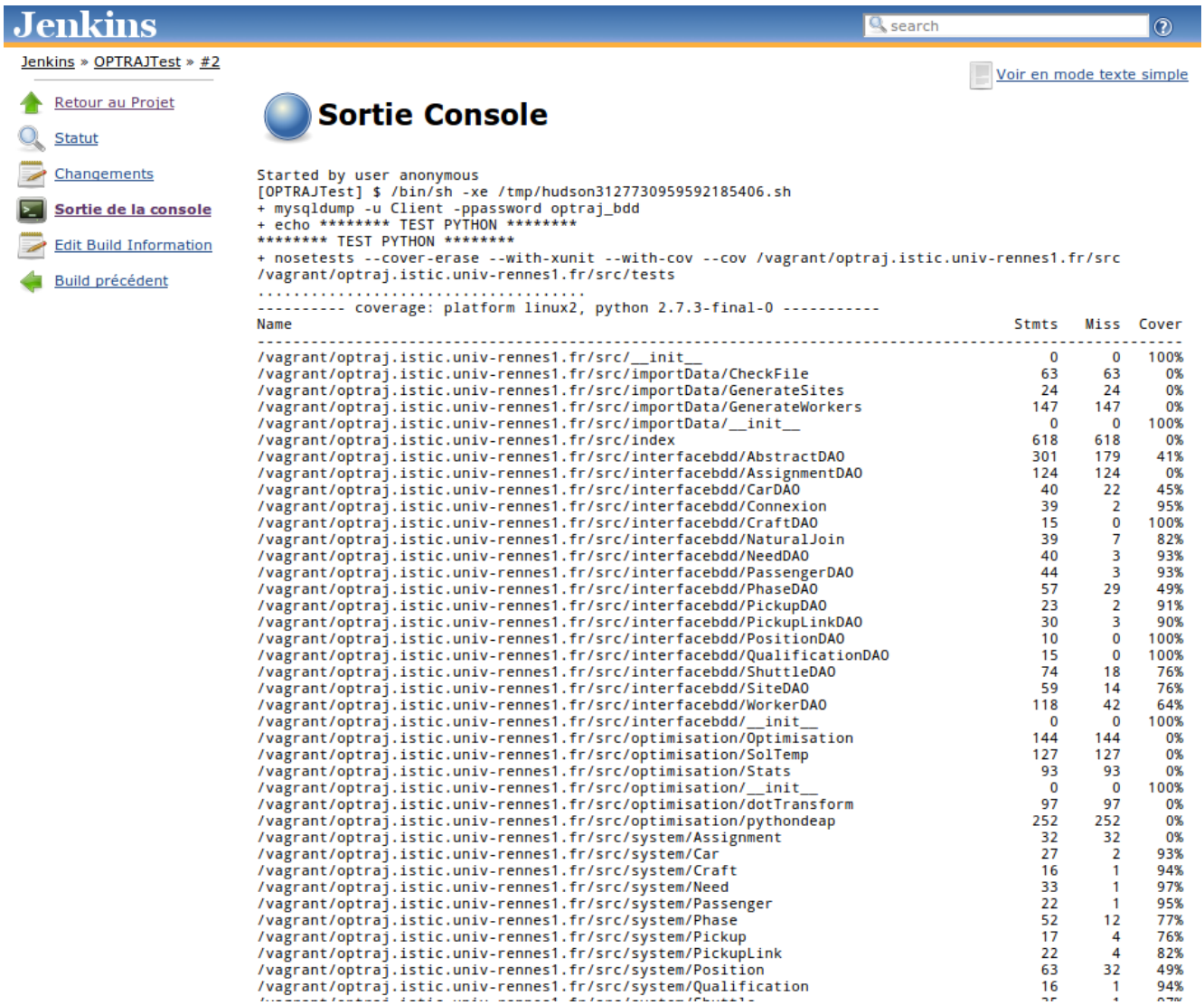
- Mémoire vive : 384 Mo
- 8 Mo de mémoire vidéo
- Disque dur 20 Go
- Une connexion internet ADSL

Coté logiciel

- un système d'exploitation 64 bits Linux, Ubuntu Precise ou Debian 7 mis à jour
- Apache 2 avec le port 80 ouvert donnant sur le dossier
  /vagrant/optraj.istic.univ-rennes1.fr/GUI/htdocs/
- PHP 5 et Curl
- Python 2.7 avec pip, flask, pymysql, pygeocoder, deap
- Mysql client 14
- Mysql server 5.5

Ensuite, depuis tout ordinateur de votre réseau via le navigateur web (nous conseillons firefox pour des raisons de compatibilité), vous pouvez vous connecter au service via http://adresse.de.la.machine/optraj.

Annexe 6 – Exemple Jenkins



Exemple de sortie Console du job Jenkins