Département Informatique

Diplôme préparé : DUT

Rapport d'alternance au Genoscope d'Evry

Cyril KEIL

Tuteur enseignant	**Maître d'apprentissage**	**Chargé de mission**
Christian MÉTAIRIE	Guillaume ALBINI Genoscope d'Evry	Bernard ALLEGRO

Année universitaire 2014-2015

Période d'apprentissage

Mémoire remis le 17 Août 2015

Remerciements

Je souhaite remercier mon maître d'apprentissage Guillaume Albini pour tout le temps qu'il m'a accordé, ainsi que pour la confiance qu'il a eu en moi pendant cette année. Je remercie également tous mes autres collègues, qui m'ont bien accueilli dans leur équipe et avec qui j'ai passé de bons moments. Enfin, merci beaucoup au CFA et au Genoscope d'Evry de m'avoir permis d'effectuer mon année d'apprentissage en leur sein, et de m'avoir donné envie de poursuivre mes études dans cette direction.

Un grand merci également à Bernard Allegro pour le temps qu'il a passé à me soutenir quand je n'avais pas d'entreprise, à me remonter le moral, et ce jusqu'en Novembre, où j'ai enfin pu commencer mon alternance au Genoscope.

Merci à mon tuteur enseignant, Christian Métairie, ainsi qu'à l'IUT d'Orsay, pour m'avoir permis d'effectuer cette formation en apprentissage. J'ai énormément appris grâce à cela, et cela me pousse à vouloir continuer mes études dans ce sens.

Enfin, je souhaite remercier mes amis qui m'ont soutenu pendant les longs mois de recherche d'entreprise, ainsi que ma famille, et j'espère qu'ils sont fiers de me voir essayer de construire un bon avenir. Merci également à Mme Bonneton pour la bonne humeur qu'elle apporte pendant ses cours, et également dans son aide pour la préparation de ce rapport.

Sommaire

Introduction

Ma mission lors de mon année d'apprentissage à l'IUT d'Orsay a été réalisée au Genoscope d'Evry, appartenant au CEA. J'ai réussi à intégrer le Genoscope d'Evry grâce à une connaissance travaillant à l'INRA, localisé dans les mêmes locaux que le Genoscope. Cette mission était pour moi très intéressante car elle permettait de joindre l'informatique, et plus particulièrement le développement web, à mon attirance pour le domaine scientifique. Ainsi, j'ai été amené à travailler sur une sous-partie du projet global de refonte de leur système de traitement des données : NGL-BI. En effet, le Genoscope a aujourd'hui recourt à de nouvelles technologies permettant de séquencer plusieurs centaines de gigabases par semaines. Cet afflux d'information à traiter pose un nouveau problème : Comment assurer le traitement des données en masse ainsi que leur suivi par les différentes équipes du Génoscope ? Afin de répondre à cette problématique, nous commencerons d'abord par faire une présentation du Genoscope ainsi que de la mission effectuée, avant de passer à une analyse plus technique de la mission et des solutions apportées pour résoudre le problème.

Présentation de l'entreprise et de la mission

Tout texte en gras possède sa définition dans le lexique, en fin de document.

Présentation de l'entreprise

J'ai effectué ma mission au sein du Genoscope, faisant partie de l'Institut de Génomique. L'institut de Génomique a été créé et intégré au CEA (Commissariat à l'énergie atomique et aux énergies alternatives) en 2007.

Le Genoscope est sur la Génopole d'Evry, et est actuellement dirigé par Jean Weissenbach. Le Genoscope compte ainsi environ 450 employés, tandis que le CEA en compte 16000, répartis dans différents domaines, comme la science du vivant, la science de la matière, l'énergie nucléaire, les applications militaires, etc. Le Genoscope fait partie depuis le 1e Mai 2007 de la section des sciences du vivant, et est formé des deux plates-formes nationales de génomique : Le Centre National de Séquençage (CNS), et le Centre National de Genotypage (CNG).

J'ai effectué ma mission sous la tutelle de mon maître d'apprentissage Guillaume Albini, chef du projet NGL (Next Generation Laboratory Information Management System), dans l'équipe de développement, séparée de la DSI. Nous sommes une dizaine de développeurs travaillant pour réaliser NGL. Nous sommes maintenant en collaboration avec l'INRA de Toulouse, avec qui nous partageons les sources de NGL.

Le Genoscope concentre donc ses activités sur le séquençage du génome, principalement d'éléments agricoles, comme le blé. L'intérêt du séquençage est multiple, mais est plus facilement parlant pour l'homme : En séquençant un gène, nous obtenons la succession des acides aminés de la ou des protéines correspondantes, mais cela sans donner leurs fonctions. L'intérêt est donc de pouvoir effectuer des comparaisons entre différents génomes afin de pouvoir isoler les protéines ainsi que leurs fonctions.

Dans le cas du cancer, par exemple, une étude est actuellement en cours par l'ICGC, dans laquelle les chercheurs cherchent à comparer les séquences de génomes extraits de cellules cancéreuses et non cancéreuses sur près de 500 patients, et ce pour 50 types de cancers étudiés. Très rapidement, des similitudes peuvent ressortir, et vont permettre d'établir une liste des mutations génomiques susceptibles d'être à l'origine de chaque type de cancer.

Cependant, un tel travail est long et très couteux, l'amélioration des machines posant des problèmes d'infrastructures, car le flot d'information peut vite poser des problèmes à une infrastructure non adaptée.

Présentation de la mission

Le Genoscope a recourt a des technologies de séquençage afin de générer des bases ADN provenant d'échantillons afin de les stocker, de les traiter et de les afficher. Le traitement des données et leur suivi par les différentes équipes du Genoscope (production des données, gestion des projets, bio-informaticiens) se faisant de façon automatisée. Cependant, l'évolution de ces technologies de séquençage haut débit ont permis d'atteindre un taux de séquençage de plusieurs centaines de gigabases de séquences en une semaine. Devant la quantité d'information à traiter, le Genoscope a jugé que les anciens outils, adaptés pour un plus faible nombre de bases, n'étaient désormais plus adaptés à ces nouvelles conditions. Il a donc été décidé de réaliser une nouvelle application, nommée NGL, afin de pouvoir répondre à de nouvelles demandes de la part des bio-informaticiens.

Afin de répondre aux nouvelles demandes, les bases de l'architecture de l'application ont été posées. Ainsi, cette application est une application web, permettant de voir l'état d'avancement des traitements, de visualiser les indicateurs de qualité sur les données, de valider ces données et de faire du reporting sur ces indicateurs. L'un des principaux buts de l'application NGL est la possibilité

L'environnement technique de l'application est le suivant (Nous reviendrons sur le pourquoi de chaque technologie abordée dans un second temps) :

Côté client, nous nous servons d'**HTML**5 et de **CSS**3, notamment avec **Bootstrap**. De plus, nous nous servons aussi de **JavaScript**, avec le **framework AngularJS**.

Côté serveur, nous nous servons de Java avec le **Framework** Play, et nous nous servons à la fois de **SQL** et de **NoSQL** pour nos bases de données, avec respectivement My**SQL** et **MongoDB**.

L'application respecte un style d'architecture nommé **REST**, tandis que le **versionnage** a été assuré dans un temps par SVN, et est maintenant assuré par Git. Afin de réaliser les différentes parties du projet et de faciliter le travail en groupe, nous nous servons de la méthode **Agile Kanban**.

Le groupe est constitué actuellement de mon maître d'apprentissage Guillaume Albini, ainsi que de huit autres développeurs, deux alternants et d'une personne se chargeant de la maîtrise d'ouvrage. C'est cette personne qui se charge de l'arrivée de ticket sur le **kanban**, que je vais présenter.

Le **Kanban** consiste en une réunion d'équipe qui s'effectue tous les matins à 10h. Celle-ci consiste en un stand-up, durant lequel chaque membre de l'équipe parle de travail effectué la veille, de ce sur quoi ils vont travailler pour aujourd'hui, des problèmes rencontrés la veille. C'est également un bon moment pour organiser une réunion avec d'autres membres de l'équipe afin de prendre des décisions affectant la suite du développement.

De façon visuelle, le **Kanban** se présente comme ceci :

Projet	BackLog	User Story	Estimation	To Do	In Progress	Done	DEV	UAT	Finish
NGL-BI					714 ** Demande				

Figure 1 : Schéma d'une ligne du **Kanban**

Chaque ligne du **Kanban** représente une partie du projet **NGL**. En effet, il existe différents projets : NGL-Réactifs, NGL-SQ, NGL-BI, NGL-SUB. Nous reviendrons plus tard sur chaque sous-projet afin d'en expliquer les tenants et les aboutissants.

Tout d'abord, chaque demande de la part des utilisateurs (ou des développeurs) est appelée un ticket. Chaque ticket est représenté sur un post-it se déplaçant sur le tableau au fil de l'avancement du développement.

Tout d'abord, le ticket arrive sur la case Backlog, où il possède un numéro de ticket, par exemple NGL-714. Sur celui-ci se trouve la demande utilisateur ainsi que sa date d'arrivée sur le tableau. En passant par les colonnes User Story et Estimation, celui-ci va se voir attribuer un développeur, ainsi qu'une difficulté allant de une à trois étoiles. Ainsi, le ticket va rester à la case To Do jusqu'à ce que le développeur se mette à travailler dessus. Au début du développement de ce ticket, il va passer dans la case In Progress jusqu'à ce que son développement soit fini. Une fois ce développement terminé, il sera mis en ligne sur un serveur de test afin de vérifier la présence de bugs, et restera donc dans les cases DEV/UAT tant que tous les bugs trouvés n'auront pas été corrigés.

L'arrivée du ticket dans la case Finish signifie la fin de son développement, ainsi que son passage futur sur le serveur de production, où les utilisateurs finaux auront à leur disponibilité la fonction rajoutée par le développement du ticket.

Enfin, certains tickets peuvent rester bloqués au « frigo », à cause d'un problème lors du développement d'un ticket (Par exemple, incompatibilité entre deux technologies qui empêchent le développement d'une fonction précise, ou des congés maternités qui bloquent le développeur qui attend des données).

Afin de gérer ces tickets et leur mise en ligne sur les différents serveurs (serveur de test, production), nous utilisons un système de versionnage. Auparavant, nous utilisions SVN pour remplir ce rôle, mais nous avons changé pour Git il y a deux mois environ. La principale cause de ce changement vient de la mise en open source du projet. En effet, nous allons collaborer avec l'INRA de Toulouse, et étant

intéressés par le travail effectué sur NGL, nous avons décidé de leur fournir le code source via GitHub afin qu'eux aussi puissent s'en servir pour effectuer du séquençage.

Ainsi, nous profitons toujours de notre système de versionnage, en ayant en plus la possibilité de rendre le code ouvert à l'INRA de Toulouse facilement. Cependant, les différences entre Git et SVN, notamment au niveau des commandes, ont posé des problèmes lors de la migration SVN -> Git. Pour l'instant, nous abordons le versionnage comme ceci :

Un développeur récupère le contenu du dépôt en ligne afin d'avoir la dernière version du projet. Il possède donc une version en local du projet. A partir de ce dépôt « local », il va créer une « branche » de son dépôt : Cela correspond à une copie identique de son dépôt local, mais sur laquelle il va pouvoir effectuer le développement du ticket. Ainsi, on peut, en créant plusieurs branches, séparer le développement de plusieurs tickets afin d'éviter les effets de bord. Une fois que le développement d'un ticket est terminé, le développeur va pouvoir « fusionner » (merge) la branche avec son dépôt local, afin que son dépôt local récupère les modifications effectuées par le développement du ticket. Une fois que le développeur est satisfait de son dépôt local, il va pouvoir envoyer les nouvelles fonctions présentes sur son dépôt local directement sur le dépôt en ligne.

Ainsi, après le développement de plusieurs fonctions, si la maîtrise d'œuvre considère que l'ensemble des développements effectués mérite une nouvelle version, et si la maîtrise d'ouvrage considère également que les fonctions rajoutées sont exemptes de bugs et remplissent bien leur fonction, on va pouvoir créer une nouvelle version de l'application qui consistera en l'état actuel du dépôt en ligne. C'est cette version qui ira sur le serveur de production et qui sera utilisée par les bio-informaticiens.

Ceci définit l'environnement dans lequel j'ai travaillé tout au long de cette année. Nous allons maintenant passer au cœur du développement de l'application qu'on m'a confiée : NGL-BI.

Analyse détaillée du projet

Le but de ma mission était la poursuite du développement d'un projet déjà entamé il y a deux ans : NGL-BI. Je devais donc rajouter de nouvelles fonctions aux modules présents à l'intérieur en fonction des tickets qui m'étaient attribués pendant le **Kanban**.

Je vais tout d'abord revenir sur un point abordé lors de la première partie : Les technologies utilisées. En effet, je vais revenir en détail sur le but de chaque technologie utilisée, pourquoi l'une et pas une autre.

Présentation des technologies

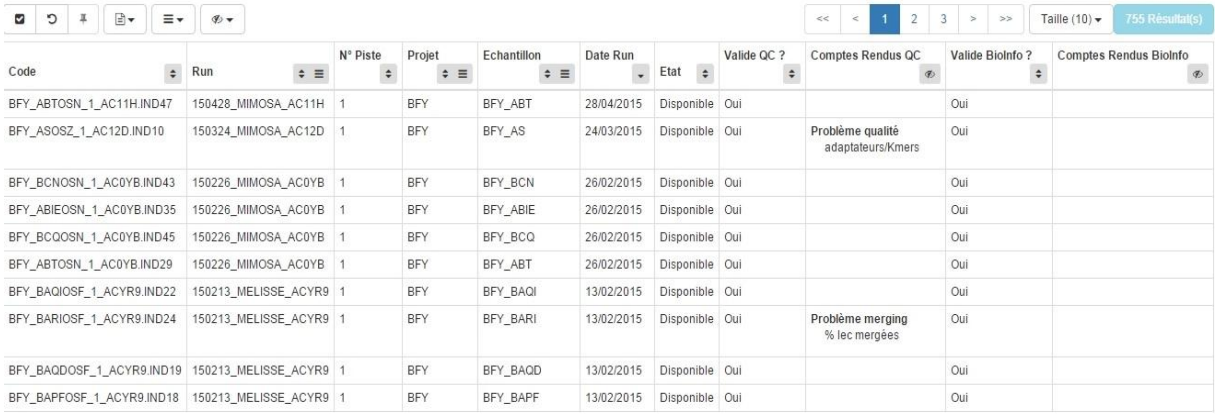
Commençons tout d'abord par le choix du type d'application. Nous avons choisi d'utiliser une base **HTML/CSS/JS** car c'était ce qui se rapprochait le plus d'une application de type bureautique. Concernant le **CSS**, nous avons choisi d'utiliser le **framework Bootstrap**, pour deux raisons : La première étant que nous ne sommes pas des designers web, **Bootstrap** nous a donc permis de passer outre la phase du design dans ce projet, tout en permettant une ressemblance entre tous les différents sous-parties du projet (NGL-BI, NGL-SQ, etc). La deuxième raison étant que l'application est vouée à être lancée sur différents supports, comme des ordinateur de bureau, ou des tablettes. Les différentes tailles d'écrans sont gérées nativement par **Bootstrap**, adaptant le contenu de la page au format de l'écran, sans avoir à fournir un travail supplémentaire. C'est ce que l'on appelle le Responsive Design.

Concernant le **JavaScript**, nous utilisons les fonctions présentes nativement dans le langage, mais nous nous servons aussi d'un **framework**, côté client donc, nommé **AngularJS**.

Ce **framework** a été choisi il a maintenant trois ans, lors du commencement du projet. A la base, c'était **jquery** qui était censé prendre ce rôle, mais lors du développement du DataTable, sur lequel je reviendrais plus tard, il a été jugé qu'il était plus simple de reproduire entièrement un nouveau Datatable entièrement avec **AngularJS**, car il présentait une meilleure évolutivité dans le temps, et surtout, **jquery** ne permettait pas, dans son datatable natif, d'effectuer de l'édition de données en masse.

L'un des autres aspects uniques d'**AngularJS** est le **two way data-binding**. Cet aspect d'**AngularJS** permet de faire le lien directement entre les données et la vue. Ainsi, quand des données sont mises à jour, la vue est également automatiquement mise à jour, et inversement.

Enfin, les directives sont également importantes dans **AngularJS**. Elles permettent de créer dans le code un **template** qui pourra être appelé partout dans le code de notre application (Pour rappel, un **template** est un morceau de code **HTML** que l'on peut insérer dans notre page, et qui aura toujours la même apparence). Le DataTable est par exemple issu d'une directive.



Code	Run	N° Piste	Projet	Echantillon	Date Run	Etat	Valide QC ?	Comptes Rendus QC	Valide BioInfo ?	Comptes Rendus BioInfo
BFY_ABTOSN_1_AC11H.IND47	150428_MIMOSA_AC11H	1	BFY	BFY_ABT	28/04/2015	Disponible	Oui		Oui	
BFY_ASOSZ_1_AC12D.IND10	150324_MIMOSA_AC12D	1	BFY	BFY_AS	24/03/2015	Disponible	Oui	Problème qualité adaptateurs/Kmers	Oui	
BFY_BCNOSN_1_AC0YB.IND43	150226_MIMOSA_AC0YB	1	BFY	BFY_BCN	26/02/2015	Disponible	Oui		Oui	
BFY_ABIEOSN_1_AC0YB.IND35	150226_MIMOSA_AC0YB	1	BFY	BFY_ABIE	26/02/2015	Disponible	Oui		Oui	
BFY_BCQOSN_1_AC0YB.IND45	150226_MIMOSA_AC0YB	1	BFY	BFY_BCQ	26/02/2015	Disponible	Oui		Oui	
BFY_ABTOSN_1_AC0YB.IND29	150226_MIMOSA_AC0YB	1	BFY	BFY_ABT	26/02/2015	Disponible	Oui		Oui	
BFY_BAQIOSF_1_ACYR9.IND22	150213_MELISSE_ACYR9	1	BFY	BFY_BAQI	13/02/2015	Disponible	Oui		Oui	
BFY_BARIOSF_1_ACYR9.IND24	150213_MELISSE_ACYR9	1	BFY	BFY_BARI	13/02/2015	Disponible	Oui	Problème merging % lec mergées	Oui	
BFY_BAQDOSF_1_ACYR9.IND19	150213_MELISSE_ACYR9	1	BFY	BFY_BAQD	13/02/2015	Disponible	Oui		Oui	
BFY_BAPFOSF_1_ACYR9.IND18	150213_MELISSE_ACYR9	1	BFY	BFY_BAPF	13/02/2015	Disponible	Oui		Oui	

Figure 2 : Exemple d'un Datatable généré automatiquement

Au final, lors du choix du **framework** à utiliser, c'est ce framwork qui a été retenu. Concernant les autres **framework**s de l'époque, **AngularJS** avait plus de fonctions que BackBone**JS**, et Ember**JS** était plus récent qu'**AngularJS**, la documentation était donc moins fournie qu'actuellement. Aujourd'hui, le choix du **framework** n'aurait peut-être pas été le même. Ces nouveaux **framework**s sont en tout cas en train de définir les nouveaux standards du web pour les autres **framework**s.

J'ai énoncé précédemment que l'application respectait une architecture **REST**. **REST** correspond à representational state transfer. Ce n'est pas un protocole ou un format, mais plutôt une marche à suivre lors de la création de l'application web. Cela permet plusieurs choses, comme la séparation client serveur. L'interface étant séparée des données, il est plus simple de faire évoluer les deux indépendamment, ce qui est ce que l'on recherche dans notre application. De plus, **REST** permet de faire du sans état, c'est-à-dire que l'ensemble des données nécessaires au serveur sont dans la requête envoyé par le client au serveur. Cela permet de ne pas avoir à traiter différents contextes lors de la réception d'une requête par le serveur, faisant ainsi gagner du temps à la fois pour le client et pour le serveur.

Chaque requête correspondant à **REST** est faite uniquement avec les verbes HTTP : GET, POST, PUT, DELETE. Cela correspond respectivement à de la récupération de données, l'insertion de données, la mise à jour de données et la suppression de données.

Enfin, chaque verbe http doit être relié à une adresse. En **REST**, on identifie chaque page **HTML** comme une ressource. Pour donner un exemple simple, imaginons que notre application gère une bibliothèque. Pour récupérer l'ensemble des livres d'un genre, nous utiliserons par exemple le verbe GET sur http://www.monsite.fr/livres/genre.

Ce genre de schéma est retrouvé très souvent dans les web services, ce qui correspond tout à fait au type d'application qu'est NGL.

Passons maintenant aux bases de données. Nous utilisons MongoDB pour le **NoSQL** ainsi que My**SQL** pour la partie **SQL**.

Pour **NoSQL**, nous avons choisi de l'utiliser car, au fil des années, les bio-informaticiens peuvent faire évoluer les séquenceurs, vouloir rajouter des données non présentes avant, retirer des données qui seraient devenues inutiles. **NoSQL** répond à la problématique « Comment conserver les anciens objets avec le nouveau modèle de données ? », car chaque document (Un document correspond à un objet avec différentes propriétés, le nom document est juste spécifique au **NoSQL**) présent dans une base **NoSQL** n'est pas obligé de suivre la même structure qu'un autre document présent dans la même base, contrairement au **SQL**. **NoSQL** va donc nous servir à stocker tous les objets ayant une structure hétérogène, comme les **readsets** générés lors du séquençage d'un génome.

Nous avons également choisi MongoDB pour nos bases **NoSQL** pour une raison simple : Étant le SGBD le plus populaire pour **NoSQL**, il était automatiquement celui le plus fourni en documentation, et il est donc simple de résoudre un problème rencontré en allant chercher la cause de son problème sur Internet.

Enfin, **SQL** sert pour tout ce qui est objets à structure fixe, mais également pour créer des contraintes d'intégrité sur nos objets. Ainsi, nous sommes sûrs que chaque propriété de l'objet respecte bien son type avant d'être inséré dans notre base **NoSQL**, car aucune vérification n'est faite côté **NoSQL**. C'est à la fois un point fort et un point faible de **NoSQL**, car cela peut à la fois accélérer le projet, mais également le ralentir si l'on se rend compte que les données présentes dans les bases de données ne sont pas adaptées a cause d'une vérification absente lors de l'insertion des données.

Il existe enfin une dernière technologie que nous utilisons, qui répond à un problème souvent posé avec les applications fréquemment utilisées : « Comment déployer une nouvelle application sans arrêter l'ancienne, de façon à ce que les utilisateurs ne soient pas stoppés dans leur travail ? ». La réponse est le load balancer. Je ne vais pas trop rentrer dans les détails, n'ayant pas pu m'en servir moi-même, mais dans les faits, il répartit la charge réseau uniquement aux serveurs capables de répondre. Ainsi, on va déployer l'application sur un serveur tandis que le load balancer renverra les requêtes en cours vers un autre serveur. Une fois l'application déployée, les requêtes reviendront vers le serveur où l'application a été déployée, tout cela rendant le déploiement transparent pour l'utilisateur.

Maintenant que nous avons défini toutes les technologies utilisées dans le projet NGL, nous allons maintenant passé au travail que j'ai effectué sur NGL-BI.

Mon travail sur NGL-BI

Premiers tickets : DataTable

Je vais présenter mon travail sur NGL-BI dans un ordre chronologique, qui correspond dans ce cas précis à une présentation dans un ordre de difficulté. En effet, j'ai tout d'abord commencé ma première semaine de travail en ne tapant aucune ligne de code, mais simplement en lisant la documentation relative à **AngularJS**, ainsi que **bootstrap** et au **framework** Play. Une fois les principes de base assimilés, j'ai eu droit à une présentation de l'architecture de NGL avant de commencer à réaliser des tickets m'étant assignés au **Kanban**. Je vais donc traiter à la fois des tickets réalisés, mais également ceux n'ayant pas pu être terminés, mais m'ayant tout de même apporté de l'expérience dans le développement.

Le premier ticket assigné consistait en un nouveau traitement pour les **readsets**. En effet, j'ai déclaré dans un premier temps les propriétés nécessaires au traitement, en attendant les données devant être insérées pour ce traitement. Cependant, je n'ai pas pu continuer ce traitement car la personne devant insérer les données est partie en congé maternité. Ce fut donc un premier ticket remis au frigo, et qui n'a à ce jour toujours pas été terminé.

Les tickets suivants concernaient dans leur majorité le DataTable (Raccourci en DT) présenté précédemment. Il est possible de grouper les données automatiquement sur le DT par colonne. Ainsi, on peut grouper des données par nom de projet, par exemple. Le but du premier ticket sur le DT était donc de pouvoir choisir entre exporter toutes les données, où alors exporter uniquement les lignes qui représentaient un groupe. Ainsi, j'ai repris la fonction d'export des données au format CSV qui existait précédemment dans le DT, en ajoutant une propriété correspondant au type d'export choisi. Je n'ai eu ensuite qu'à récupérer les lignes groupées dans le DT pour les insérer dans le fichier CSV.

Le ticket suivant concerne aussi l'export CSV. Dans le DT, il est possible de cacher des colonnes si l'utilisateur les considère comme inutiles actuellement. Le but de ce ticket était de ne pas exporter les colonnes cachées par l'utilisateur. Pour ce faire, j'ai, pour chaque colonne, regardé si l'attribut « hide » de la colonne était à vrai ou faux, et dans le deuxième cas, je gardais cette colonne pour l'export.

Une autre demande au **Kanban** concerne un bouton permettant de regrouper toutes les lignes en une seule. Le but est surtout d'avoir la somme totale d'une propriété sur un projet donné. Pour faire cela, au lieu d'envoyer à ma fonction le nom de la colonne sur laquelle on groupe, on envoie plutôt un nom de code comme « all ». Il faut ensuite faire en sorte que la fonction sélectionne toutes les lignes du tableau pour faire les sommes et les moyennes, et pas seulement certaines lignes correspondant à certains groupes, comme c'était le cas auparavant.

Les deux tickets suivant sont légèrement plus compliqués que ceux présentés auparavant. En effet, c'est avec ceux la que j'ai commencé à vraiment me servir d'**AngularJS** et de **Bootstrap**.

Le premier était de générer automatiquement un menu déroulant avec **Bootstrap**, qui contiendrait toutes les colonnes sur lesquelles il était possible d'effectuer un groupement, puis en fin de liste donner la possibilité de grouper toute la sélection.

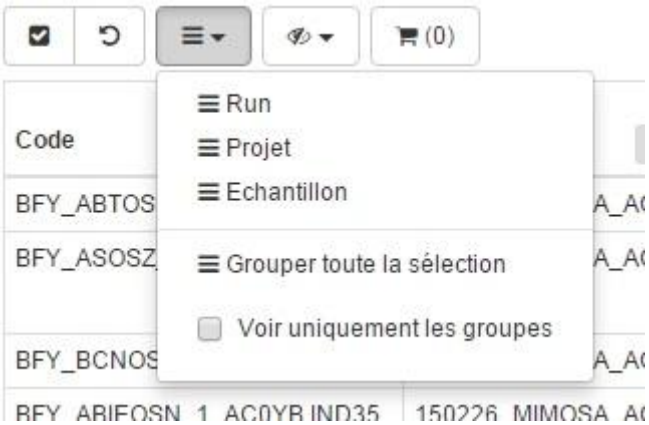


Figure 3 : Exemple du menu déroulant permettant d'effectuer un groupement sur le DataTable

(La case « Voir uniquement les groupes » n'est pas de ma réalisation)

Lors de la génération du Datatable, j'ai rajouté du code au niveau de la génération des boutons au dessus du datatable, permettant de créer un menu déroulant. Pour remplir ce menu, une fonction va rajouter dans une liste toutes les colonnes dont l'attribut « group » est à vrai, signifiant qu'elles peuvent être groupées (Et non pas qu'elles sont déjà groupées). Ensuite, **AngularJS** va permettre de générer du code **HTML** en boucle pour rajouter une à une chaque colonne dans notre menu déroulant provenant de notre fonction. Le bouton « Grouper toute la sélection » est quant à lui toujours afficher si le groupement sur tout le tableau est jugé possible. Toujours sur le même ticket, j'ai effectué le même traitement pour avoir toutes les colonnes pouvant être cachées.

Le deuxième ticket concerne les listes déroulantes présentes avant la génération du DT :

Figure 4 : Listes pour la génération du DataTable

Prenons pour exemple la première liste, en haut à gauche, celle permettant de choisir des projets. On peut rentrer dans la barre le nom du projet que l'on cherche, ou juste une partie du nom. La liste déroulante va donc sortir plusieurs lignes correspondantes à notre recherche, et que nous pourrons sélectionner. Le but du ticket est de faire en sorte que toute ligne cochée soit conservée dans la liste déroulante en permanence, au dessus du résultat de recherche, afin de pouvoir décocher facilement un projet sans avoir à refaire une recherche pour le supprimer.

Pour cela, on va procéder de la même façon que pour les colonnes pouvant être groupées : On va regarder dans les éléments de la liste ceux qui ont déjà été cochés, puis on va les rajouter dans la liste avec un **template HTML**, tout cela grâce à **AngularJS**.

```
// Liste des items déja cochés
+'<li ng-repeat-start="item in getSelectedItems()" ng-if="groupBy(item, $index) && acceptsMultiple()"></li>'
    +'<li class="dropdown-header" ng-if="groupBy(item, $index)" ng-bind="itemGroupByLabel(item)"></li>'
+'<li ng-repeat-end ng-if="item.selected" ng-click="selectItem(item, $event)">'
    +'<a href="#">'
        +'<i class="fa fa-check pull-right" ng-show="item.selected"></i>'
        +'<span class="text" ng-bind="itemLabel(item)" style="margin-right:30px;"></span>'
    +'</a></li>'
+'<li ng-show="getSelectedItems().length > 0" class="divider pull-left" style="width: 100%;"></li>'
```

Figure 5 : Code permettant de récupérer les éléments déjà cochés

Pour expliquer la base d'**AngularJS**, le code ci-dessus va nous servir : Le ng-repeat sert à boucler sur une liste d'éléments, ici récupérés avec la fonction getSelectedItems(). On va donc récupérer les items, puis pour chaque item mettre son nom et une icône pour signifier qu'il est coché. Enfin, avec la fonction selectItem(), on va pouvoir relier ce qui à la base n'est que du texte à une fonction, permettant de décocher l'élément choisi lors d'un clic dessus. Au final, le fonctionnement est le même que dans une liste déroulante normale, sauf qu'ici, on rassemble des propriétés déjà cochées.

Les statistiques préconfigurées

Dans NGL-BI, l'accent est porté sur la consultation est la modification des données présentes grâce au séquençage d'un génome. Afin de pouvoir consulter ces données, il nous est possible de générer des tableaux à l'aide de notre module DataTable, mais il est également possible de générer automatiquement des graphiques en fonction des données récoltées. Cependant, dans un souci de temps, il m'a été demandé au **Kanban** de permettre de choisir des préconfigurations pour les statistiques, afin que les utilisateurs puissent consulter facilement les stats d'un projet, comme le projet blé, mais cela doit être possible avec d'autres projets si les bio-informaticiens le demandent.

L'intérêt de ce ticket est de pouvoir toucher à la fois du côté client à la génération de graphiques avec Highcharts, mais également de mieux comprendre le fonctionnement interne de l'application NGL-BI en réussissant à faire le lien entre le **JavaScript**, l'api récupérant les données, et ma base de données **NoSQL**.

Ainsi, après quelques essais où j'ai fait fausse piste, en me servant de choses inutiles, j'ai réussi à avoir un résultat satisfaisant.

Tout d'abord, il a fallu que je créé une api permettant de ramener mes données. Pour cela, un système CRUD était nécessaire avec notre système **REST**. Ainsi, j'ai défini sous quel format devait être les données que je voulais récupérer, contenant donc le nom du projet, les colonnes voulues pour les insérer dans le DataTable, une expression régulière pour pouvoir séparer les différents sous-parties du projet blé. Tout cela s'effectue en java, côté serveur.

Une fois ce système créé, il a fallu insérer les données dans MongoDB. Ici, tout est fait en **JSON**, ce qui facilite la lecture, l'insertion et la modification des données. Pour insérer les données dans la base de données, et aussi pour les récupérer, il faut créer des routes compréhensibles pour le **framework** Play. Ainsi, comme dit précédemment pour **REST**, on se sert des verbes GET, POST, etc, en reliant chaque verbe à une adresse afin de pouvoir stocker les données sur la base de données.

On a donc pour les stats ces routes :

```
#stats api
GET     /api/stats/configs                  controllers.stats.api.StatsConfigurations.list()
GET     /api/stats/configs/:code            controllers.stats.api.StatsConfigurations.get(code)
POST    /api/stats/configs                  controllers.stats.api.StatsConfigurations.save()
PUT     /api/stats/configs/:code            controllers.stats.api.StatsConfigurations.update(code)
DELETE  /api/stats/configs/:code            controllers.stats.api.StatsConfigurations.delete(code: java.lang.String)
```

Figure 6 : Exemple de routes pour la récupération et l'insertion de données pour les stats

Une fois les données créées, il faut maintenant les traiter quand elles arrivent côté client. De base, le menu se présente comme ceci :

Figure 7 : Présentation du menu des stats

Nous arrivons d'abord sur le menu « Sélectionner les **Readsets** », sur lequel nous avons le même menu que la figure 4. Un DataTable est généré, puis une fois que nous sommes satisfaits des données sélectionnées, on rajoute notre sélection au panier. Une fois ceci fait, nous allons dans le menu « Configuration Statistiques », afin de choisir les propriétés pour lesquelles nous voulons afficher un graphique. Il est possible d'avoir des données brutes, ou alors de mesurer l'écart type pour toutes les données d'une colonne.

Ceci correspond à une configuration manuelle. Dans mon cas, je devais faire en sorte de ne pas avoir à choisir le nom du projet, ou les propriétés à afficher, mais simplement choisir le nom de la configuration dans une liste déroulante.

Ces configurations sont récupérées avec l'api créée précédemment. Pour le traitement, nous devons donc vérifier dans quel cas nous nous trouvons : Soit la configuration est faite manuellement, soit elle provient de la base de données. Dans le deuxième cas, nos données sont récupérées via le service $http fourni par **AngularJS**, et sa méthode get, sur la route créée précédemment. Une fois ces données récupérées, nous générons un DataTable contenant toutes les colonnes précisées dans notre modèle de données, écrit en **JSON** sur notre base de données.

Enfin, nous allons, pour chaque propriété, voir le type de graphique qui est demandé : Soit des valeurs brutes, soit l'écart type pour chaque **readset** sur cette propriété. Nous avons donc deux fonctions, très simples à créer, étant donné qu'Highcharts, notre plugin permettant de générer les graphiques, n'a besoin en entrée que d'un objet **JSON** afin de générer le graphique directement sur la page.

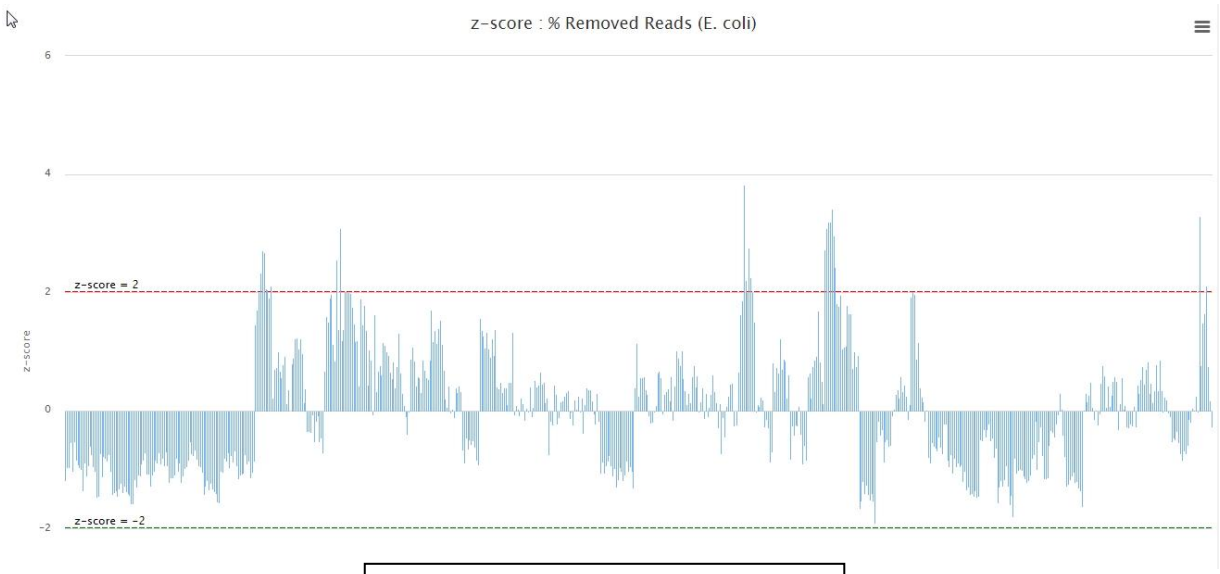


Figure 8 : Graphique généré automatiquement avec Highcharts

Ici, nous avons affaire à un graphique affichant l'écart type de chaque **readset** sur le pourcentage de reads enlevés lors du séquençage du **Readset**.

L'un des autres aspects intéressant d'Highcharts est de pouvoir, pour chaque barre de notre histogramme, attribuer un lien vers un nouvel onglet, qui s'affichera en cliquant dessus. Ainsi, j'ai pu facilement relier chaque barre de cet histogramme automatique à la page détaillée de son **Readset**, car tout cela est fait via du simple **JSON** :

```
point : {
        events : {
                // Redirects to valuation page of the clicked readset
                click : function() {
                $window.open(JSRoutes.controllers.readsets.tpl.Readsets.get(this.name).url);
                }
        }
},
```

Ici, pour chaque point de l'histogramme, on utilise le service $window d'**AngularJS** sur une route prédéfinie dans notre api. La fonction $window.open permet d'ouvrir un nouvel onglet avec un lien précisé en paramètre.

Concernant nos graphiques, il est également possible de les exporter, mais je reviendrai dessus après le ticket m'ayant pris le plus de temps à réaliser : Le bilan de production.

Le bilan de production

Au Genoscope, il est important de pouvoir avoir accès au bilan de production à des fins statistiques. On m'a ainsi demandé de réaliser un bilan de production accessible via NGL-BI.

Ce bilan devait répondre à plusieurs critères : Il devait tout d'abord permettre de consulter un bilan général du nombre de bases générées sur toutes les années, mais également d'autres types de bilans : Un bilan mensuel et trimestriel par année du nombre de bases générées, un bilan annuel du nombre de bases générées par type de séquenceur, un bilan du nombre de bases générées et les pourcentages associés pour les 10 plus gros projets de l'année, et enfin un bilan annuel du nombre de bases générées par type de sous-projet.

La création d'une page entière dans NGL-BI a été plutôt stimulante. En effet, j'ai dû effectuer toutes les étapes pour poser les bases de ma page avant même de pouvoir commencer à travailler sur le bilan en soi.

J'ai donc commencé par quelque chose de simple en ajoutant la page dans le menu de l'application. Ensuite, j'ai rajouté les routes permettant d'accéder aux différentes pages de mon bilan : La route permettant d'effectuer du reverse **routing** avec Play, ainsi que la route pour le bilan de chaque année, et celle pour le bilan général. J'ai d'ailleurs eu quelques problèmes avec mes routes en essayant de les faire fonctionner, car une de mes routes amenait sur elle-même, et provoquait donc un chargement infini de la page.

Le fonctionnement du **routing** est le suivant : l'application analyse l'URL lorsque l'on arrive sur la page. Pour chaque URL, nous avons un **template** qui est chargé via une route définie dans le fichier de configuration, ainsi qu'un contrôleur associé. Si l'URL fournie ne correspond à aucune URL

connue, il est possible de fournir à l'application une route par défaut afin d'afficher autre chose qu'une page d'erreur.

Une fois mes routes en place, et mes pages accessibles, j'ai commencé la réalisation de mon bilan. La première étape est de récupérer les données. Pour cela, nous nous servons toujours du service $http fourni par **AngularJS**, en précisant la route par laquelle aller chercher les données. Cependant, cela a fait apparaitre un premier problème : La récupération complète des **readsets**, contenant les données nécessaires à notre bilan, fait très vite augmenter la quantité récupérée au dessus du gigaoctet, finissant par faire planter l'onglet courant du navigateur. Heureusement, il est facile de préciser simplement quelles propriétés nous voulons récupérer et lesquelles nous voulons ignorer, faisant rapidement passer le volume des données de 1 Go à 40 Mo. Une fois ce problème réglé, j'ai également fait en sorte de récupérer les données globales de nos **runs** ainsi que de nos projets. C'est à partir de ces 3 groupes d'objets que j'ai fabriqué mon bilan.

Il a ensuite fallu décider du rendu que mon bilan devait avoir. Je me suis donc inspiré de ce que proposait **Bootstrap**, ainsi que de ui.bootstrap, une version fonctionnant avec **AngularJS**. Le rendu final de chaque partie du bilan était donc le suivant :

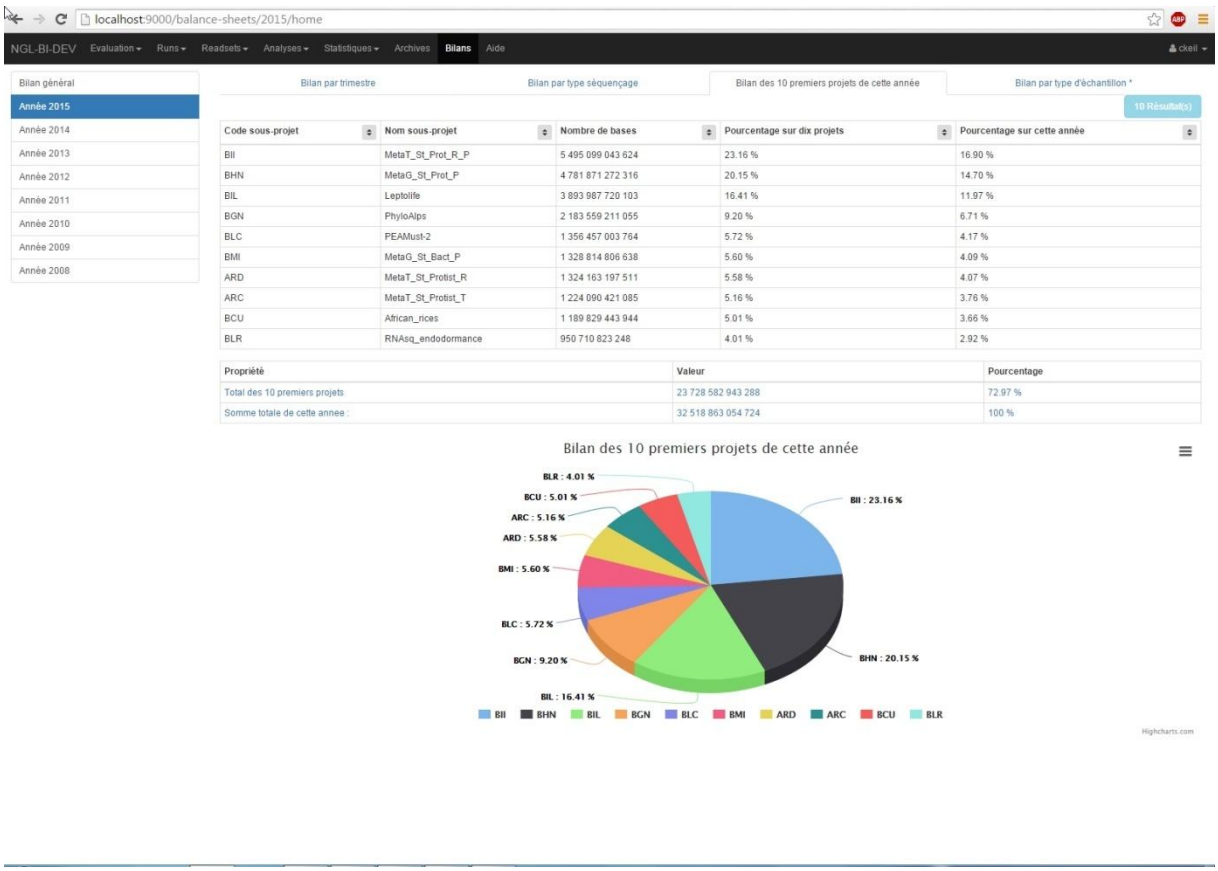


Figure 9 : Capture d'écran d'un onglet du bilan de production

Ainsi, en haut à gauche du bilan, nous pouvons apercevoir le même système de sélection que dans la figure 7, pour les statistiques. La séparation du bilan général et des bilans annuels devait se faire, car le chargement du bilan général pouvant prendre plusieurs dizaines de secondes alors que le chargement d'une seule année ne prend que quelques secondes. Cela rendait le chargement long inutilement.

L'utilisateur choisit donc entre le bilan général, ou le bilan d'une année précise. Pour le bilan général, peu de choses à dire : Un simple DataTable affichant le nombre de bases générées chaque année, ainsi qu'un histogramme illustrant notre DataTable. Le bilan annuel est plus intéressant, car plus complet dans la variation des formes adoptées pour illustrer chaque partie du bilan. Ainsi, on peut avoir plus d'un DataTable par onglet du bilan, et ce ne sont pas toujours des histogrammes qui représentent ce DataTable. Mais je reviendrai plus tard sur la nécessité d'avoir plusieurs tableaux.

Une fois le design de chaque onglet défini, j'ai donc commencé à traiter les données reçues (Nous ne nous attarderons que sur le bilan annuel, le bilan général n'étant qu'une version simplifiée de tout le système utilisé dans le bilan annuel).

Concernant le premier bilan, à savoir le bilan mensuel et trimestriel, j'ai tout d'abord créé un objet ayant pour modèle de données :

-trimestre: Entier
-mois: Chaîne de caractères
-nombre de bases: Entier

J'ai ensuite mis dans une liste autant d'objets qu'il y avait de mois écoulés dans cette année, afin de ne pas afficher les mois dans l'année actuelle où il n'y avait aucune donnée.

La suite est ensuite simple : On parcourt chaque **readset**, et on ajoute son nombre de bases générées au mois correspondant dans notre liste d'objets. Tout ceci nous donne une somme pour chaque mois, et nous allons donc pouvoir remplir les lignes de notre DataTable.

Mais avant de le remplir, il faut aussi calculer la somme pour chaque trimestre, et insérer la ligne calculée à la bonne position, sinon quoi nous aurions un DataTable sans lien entre les lignes des mois et les lignes des trimestres. Une fois ceci fait, nous avons un rendu final ressemblant à ceci pour cette année :

Trimestre	Mois run	Nombre de bases
1	Janvier	4 654 931 584 758
1	Février	5 487 119 904 694
1	Mars	5 605 559 026 896
	Somme :	15 747 610 516 348
2	Avril	4 965 878 240 080
2	Mai	2 926 416 591 474
2	Juin	5 644 748 011 150
	Somme :	13 537 042 842 704
3	Juillet	3 234 209 695 672
	Somme :	3 234 209 695 672
	Somme totale de cette annee :	32 518 863 054 724

Figure 10 : Rendu d'un DataTable pour le bilan mensuel d'une année

Le DataTable nous permet aussi d'attribuer des styles à une ligne en particulier, c'est comme cela que nous pouvons, par exemple, attribuer une classe de **Bootstrap** colorant le texte en bleu à une ligne importante, comme la somme pour un trimestre, ou la somme totale pour cette année.

Les autres onglets possèdent eux un deuxième DataTable à cause d'un problème lié à la fonction de tri. En effet, sur le bilan mensuel, nous n'avons pas jugé nécessaire d'insérer la fonction de tri sur un bilan déjà trié par mois. Cependant, sur les autres, les utilisateurs peuvent vouloir trier sur le type de séquenceur, ou sur le nombre de bases directement. Le problème soulevé est le suivant : Comme nous pouvons le voir sur la figure 10, les lignes pour les sommes font aussi partie du DataTable. Si nous nous servons de la fonction de tri, leur position va alors changer. Comment permettre de trier les données tout en conservant un bon affichage pour les sommes ? La réponse la plus facile à mettre en œuvre est de créer un deuxième DataTable, affichant les données comme le pourcentage de bases des 10 plus gros projets de cette année par rapport à la somme totale du nombre de bases pour cette année, ou alors la somme totale du nombre de bases sur l'année sélectionnée.

Code sous-projet	Nom sous-projet	Nombre de bases	Pourcentage sur dix projets	Pourcentage sur cette année
BII	MetaT_St_Prot_R_P	5 495 099 043 624	23.16 %	16.90 %
BHN	MetaG_St_Prot_P	4 781 871 272 316	20.15 %	14.70 %
BIL	Leptolife	3 893 987 720 103	16.41 %	11.97 %
BGN	PhyloAlps	2 183 559 211 055	9.20 %	6.71 %
BLC	PEAMust-2	1 356 457 003 764	5.72 %	4.17 %
BMI	MetaG_St_Bact_P	1 328 814 806 638	5.60 %	4.09 %
ARD	MetaT_St_Protist_R	1 324 163 197 511	5.58 %	4.07 %
ARC	MetaT_St_Protist_T	1 224 090 421 085	5.16 %	3.76 %
BCU	African_rices	1 189 829 443 944	5.01 %	3.66 %
BLR	RNAsq_endodormance	950 710 823 248	4.01 %	2.92 %

Propriété	Valeur	Pourcentage
Total des 10 premiers projets	23 728 582 943 288	72.97 %
Somme totale de cette annee :	32 518 863 054 724	100 %

Figure 11 : Exemple d'un double DataTable pour un onglet du bilan

Comme je l'ai annoncé précédemment, chaque onglet du bilan possède son propre graphique associé, généré avec Highcharts. J'ai donc pu découvrir un peu plus en profondeur ce plugin et les différentes options de rendu possibles, comme les diagrammes en camembert, en 3D, et tout cela très facilement paramétrable grâce au **JSON**.

J'ai également, au détour d'un ticket parallèle à celui-ci, réalisé l'export de mes graphiques. Tout cela est possible avec un module compatible avec Highcharts, qui propose à l'utilisateur de choisir soit d'imprimer le graphique, soit de l'exporter dans différentes formats, comme JPG, PNG, PDF, SVG. Il est bien entendu possible de forcer le type d'export, mais cela n'aurait aucun sens ici, car nos utilisateurs finaux ont des besoins qui changent en fonction de ce qui peut leur être demandé.

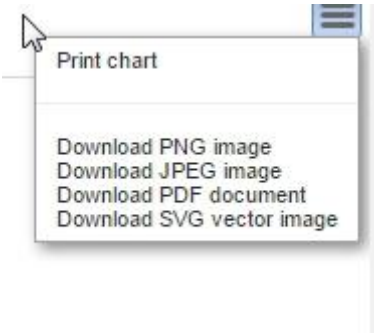


Figure 12 : Exemple du menu généré pour l'export d'un graphique

Cependant, même ce module n'est pas exempt de bugs, et l'export des histogrammes montre un problème : Si une barre de l'histogramme est de couleur noire, celle-ci ne se verra pas sur le fichier exporté. Heureusement, il est toujours possible de régler le problème en choisissant les couleurs des barres de l'histogramme en avance.

Je n'ai également pas échappé aux bugs pendant la création de mon bilan. L'un des plus préoccupants vient de l'accumulation des données lors du changement d'année. Si l'on passe par les bilans de 2015, 2014, et encore 2015, sans attendre que ceux-ci aient fini de charger, les données vont s'accumuler deux fois sur l'année 2015, ce qui faussera le bilan. J'ai réussi à résoudre ce bug en regardant lors de ma réception des données si des **readsets** n'étaient pas déjà présents pour cette année, et si oui, je supprimais intégralement la liste de mes **readsets** déjà présents pour les remplacer par ceux reçus à l'instant.

J'ai également dû régler des bugs dus à la conception de Highcharts. En effet, si un graphique est généré dans une zone de la page sans pouvoir récupérer la taille de cette zone, le graphique va automatiquement prendre une taille de 600*400. Le problème est que je me sers d'un système d'onglets pour mon bilan, et tout onglet non sélectionné voit sa zone possédant une taille non définie. Or, au début, je générais tous mes graphiques dès que l'utilisateur cliquait sur une année. Cela veut dire que tous les graphiques, en dehors de celui affiché en premier lors du chargement de l'année, auront une taille de 600*400, ce qui compressera notre graphique et le rendra illisible. Pour pallier à ce bug, j'ai fait en sorte que le graphique ne soit chargé qu'au moment où l'utilisateur clique sur l'onglet du bilan souhaité, ce qui est transparent pour l'utilisateur car il est généré trop vite pour créer une gêne.

J'ai donc appris énormément de choses grâce à ce bilan de production, que ce soit pour la création de mes routes, ou alors la gestion de mes contrôleurs et de mes modèles avec **AngularJS**, des fonctionnalités avancées avec Highcharts, comment gérer des bugs créés par un conflit entre deux technologies, etc.

Conclusion professionnelle

Le bilan de mon expérience professionnelle est positif. En effet, j'ai pu travailler avec des technologies vraiment nouvelles, et qui sont très demandées sur le marché de l'emploi, ce qui va me permettre de faciliter ma recherche. Dans mon travail, j'ai apprécié la façon dont chaque développeur pouvait gérer sa charge de travail, mettre des priorités sur un développement ou un autre. Travailler en se sentant libre de ses choix apporte vraiment un changement bienvenu par rapport à l'IUT. Enfin, j'ai également apprécié tous les moments entre les membres de l'équipe, les repas, les restaurants, qui vont sans doute me manquer l'année prochaine.

Conclusion personnelle

Cette expérience m'a personnellement plu à cause du sentiment de nouveauté ressenti lors de mon arrivé en entreprise. On découvre de nouvelles technologies qui ne sont pas enseignées en cours, même si cela varie en fonction de l'entreprise dans laquelle on effectue notre mission. Je pense que l'IUT devrait effectuer un audit des technologies en vogue ces derniers temps afin d'avoir une idée des sujets à aborder afin d'apporter des éléments concrets au CV de l'étudiant lors de sa sortie de DUT. Personnellement, j'ai eu la chance d'avoir un maître d'apprentissage qui tentait de me pousser à avoir une réflexion d'ingénieur, et je pense que cela me servira lors de ma poursuite d'études à l'EFREI, où je resterai en alternance.

Lexique

- Agile : Regroupement de méthodes permettant la bonne gestion du développement d'un projet informatique, permettant une grande réactivité lors d'un problème pendant le développement d'un projet.
- AngularJS : Framework MVC libre et open-source développé par Google depuis 2009, fondé sur l'extension du langage HTML. Voir aussi : two-way data binding
- Bootstrap : Framework CSS développé par Twitter depuis 2011, et contenant une vaste collection d'outils permettant de faciliter la réalisation du design d'un site web, allant au placement en grille du contenu jusqu'à la gestion de la totalité des types d'écrans (smartphones, ordinateurs portables, etc).
- CSS : Langage permettant de décrire l'agencement et l'apparence des éléments d'un document HTML et XML.
- HTML : Format de données conçu pour représenter les pages web. C'est un langage de balisage permettant de structurer ses pages et de les mettre en forme, notamment avec du CSS.
- JavaScript : Langage de programmation de scripts utilisé principalement dans les pages web interactives, mais aussi pour les serveurs.
- JSON : Format de données textuelles prenant racine dans la notation des objets du javascript, permettant de représenter des informations de façon structurée, comme le fait XML.
- Kanban : Méthode agile pour le développement de projets informatiques, où la progression d'une nouvelle fonction se fait via l'avancement d'un ticket à travers différents états, comme « To do » ou « Done ».
- NoSQL : Not Only SQL, désigne une catégorie de SGBD, Système de Gestion de Bases de Données, n'étant pas fondée sur l'architecture classique des bases relationnelles, car le contenu de la table n'est plus obligé de suivre un schéma répété pour chaque élément. On y laisse ainsi quelques fonctionnalités au profit de la simplicité.
- MongoDB : Système de gestion de bases de données basé sur NoSQL, le plus populaire de sa catégorie.
- Readset : Partie séquencée d'un genome, contenant dans notre cas plusieurs centaines de milliers de bases ADN par Readset. Un ensemble de readsets forme un run.
- Routing : Action permettant de relier une adresse du navigateur à un template et des données.
- Run : Ensemble des readsets générés par le séquenceur lors d'une session de séquençage d'un genome. Un projet de séquençage est ainsi constitué de plusieurs runs.
- SQL : Langage servant à exploiter des bases de données relationnelles, en opposition à NoSQL, où la notion de table est importante, et où chaque élément de la table doit respecter le format de données imposé.
- Template : Morceau de code générique permettant l'implémentation de données en son sein sans considération du type de données. Dans notre cas, un template va nous permettre par exemple de définir comment est formé le DataTable, les boutons associés. C'