



UNIVERSITÉ DE
MONTPELLIER



Rapport de stage

Amélioration de l'outil « Scrum Companion » et création du « CampusPlaylist »

AUTRICE

Stella-Maria RENUCCI

TUTEURS

M. Sergio MIRANDA CARVALHO, Scrum Master
M. Antoine CHOLLET, Professeur

Diplôme de BUT Informatique
Année universitaire 2022-2023

Remerciements

Je tiens à exprimer ma profonde gratitude envers toutes les personnes qui m'ont soutenu et accompagné tout au long de ce stage.

Tout d'abord, je tiens à remercier sincèrement mon tuteur professionnel, M. Sergio MIRANDA. Son soutien et son aide ont grandement contribué à mon apprentissage et développement professionnel. J'ai grandement apprécié travailler avec lui et je le remercie d'avoir été mon client et tuteur durant ce stage, ses projets étaient très intéressants et enrichissants.

Je tiens également à remercier mon tuteur enseignant, M. Antoine CHOLLET, pour son suivi pédagogique et son aide concernant la rédaction de ce rapport ou encore, les différentes questions que j'ai pu avoir au cours de ce stage.

Un grand merci notamment à l'entreprise GoodBarber sans qui ce stage n'aurait pas été possible, collaborer avec vous était une expérience inoubliable. Par ailleurs, je souhaite remercier particulièrement M. Joel CARLI pour s'être occupé de mon dossier au sein de l'entreprise.

Enfin, j'aimerais exprimer ma reconnaissance au corps enseignant et à la direction de l'IUT de Montpellier-Sète. Notamment à Mme Manon FRANCOIS qui a su répondre à mes interrogations concernant le déroulement du stage et m'a aidé pour l'extension de celui-ci.

Résumé

L'application « Scrum Companion » est un outil d'aide à la gestion Scrum, permettant notamment au Scrum Master de planifier ses différents sprints, de générer des statistiques concernant ceux-ci ou encore, de mieux gérer ses équipes. Cet outil a besoin de nouvelles fonctionnalités, telles qu'un système de désignation d'un référent technique de projet ou encore, une intégration à l'application Slack. Il nécessite également une amélioration en termes d'UX.

L'application « CampusPlaylist », quant à elle, est un jeu interne où les collaborateurs proposent une musique qui sera ajoutée à une playlist. Par la suite, les différents joueurs doivent associer chaque musique au collaborateur qui l'a proposé. Il existe un système de points, de gagnants et de perdants.

Ces deux applications ont été développées en utilisant le Framework Django (Python), différentes librairies JavaScript et CSS, telles que JQuery ou Bootstrap. Le « Scrum Companion » utilise les API Jira et Slack tandis que le « CampusPlaylist » utilise l'API Spotify ainsi qu'une API REST interne.

Mots-clés: Scrum, Spotify, Framework Django, Slack, librairies, Jira, API, REST.

Abstract

The "Scrum Companion" app is a Scrum assisting tool, allowing the Scrum Master to plan his various sprints, to generate statistics about them or to manage his teams more efficiently. This tool needs new features, such as a system to designate a technical assignee or an integration with the Slack application. It also needs improvements in terms of UX and UI.

Meanwhile, the "CampusPlaylist" app is an intern game where the collaborators choose a song that will be added to a playlist. Then, players will have to guess who added which song to the playlist. A point system, with winners and losers, exists.

These two apps were developed using the Django Framework (Python), various JavaScript and CSS libraries, such as JQuery or Bootstrap. The "Scrum Companion" app uses Jira and Slack APIs, and the "CampusPlaylist" app uses Spotify API and a REST API.

Key words: Scrum, Spotify, Framework Django, Slack, libraries, Jira, API, REST.

Sommaire

INTRODUCTION.....	1
1 CONTEXTE DU STAGE	2
1.1 CONTEXTE	2
1.2 PRÉSENTATION DE L'ENTREPRISE.....	2
1.3 BESOINS ET MISSIONS.....	4
1.3.1 <i>Projet « Scrum Companion ».....</i>	4
1.3.2 <i>Projet « CampusPlaylist ».....</i>	5
2 PROJET « SCRUM COMPANION ».....	6
2.1 ANALYSE DE L'ENVIRONNEMENT TECHNIQUE	6
2.1.1 <i>Problèmes initiaux</i>	6
2.1.2 <i>Analyse des technologies</i>	7
2.1.3 <i>Architecture du projet</i>	8
2.1.4 <i>Architecture de la base de données</i>	10
2.1.5 <i>Application et fonctionnalités</i>	11
2.2 SPÉCIFICATIONS FONCTIONNELLES ET TECHNIQUES.....	12
2.3 RAPPORT TECHNIQUE.....	14
2.3.1 <i>Application « Project Writing »</i>	14
2.3.2 <i>Preview de Review</i>	29
2.3.3 <i>Amélioration de la Review</i>	31
2.3.4 <i>Intégration Slack</i>	39
2.4 MANUEL D'UTILISATION	42
2.4.1 <i>Application « Project Writing »</i>	42
2.4.2 <i>Preview de Review</i>	43
2.4.3 <i>Amélioration de la Review</i>	43
2.4.4 <i>Intégration Slack</i>	43
3 PROJET « CAMPUSPLAYLIST ».....	44
3.1 SPÉCIFICATIONS FONCTIONNELLES ET TECHNIQUES.....	44
3.2 RAPPORT TECHNIQUE.....	47
3.2.1 <i>Maquette du projet</i>	47
3.2.2 <i>Architecture du projet</i>	48
3.2.3 <i>Architecture de la base de données</i>	49
4 MÉTHODOLOGIE ET ORGANISATION.....	52
4.1 ORGANISATION DU TRAVAIL.....	52
4.2 MÉTHODES DE TRAVAIL.....	53
4.3 RÉALISATIONS MINIMES.....	53
CONCLUSION.....	54

Table des figures

FIGURE 1 : ARCHITECTURE DU PROJET.....	8
FIGURE 2 : ARCHITECTURE EXTERNE DU PROJET.....	10
FIGURE 3 : LISTE DES FONCTIONNALITÉS DU SCRUM COMPANION	11
FIGURE 4 : EXEMPLE DE CODE POUR LE TABLEAU DE CHARGE COURANTE	16
FIGURE 5 : REQUÊTE À L'API JIRA POUR AVOIR LES PERSONNES ASSIGNNÉES AUX PROJETS.....	16
FIGURE 6 : PREMIÈRE VERSION DU TABLEAU DE CHARGE COURANTE.....	17
FIGURE 7 : REQUÊTE JIRA POUR OBTENIR LES RÉSULTATS DE LA CHARGE COURANTE	18
FIGURE 8 : CODE RÉCUPÉRANT LES POINTS D'EFFORTS D'UN PROJET.....	19
FIGURE 9 : VERSION FINALE DU TABLEAU DE CHARGE COURANTE	20
FIGURE 10 : REQUÊTE FINALE POUR LES DONNÉES DES TABLEAUX	21
FIGURE 11 : EXTRAIT DU CODE POUR FILTRER LES ÉQUIPES	23
FIGURE 12 : EXEMPLE DE CLASS-BASED VIEWS	24
FIGURE 13 : PREMIÈRE VERSION DE LA RÉCUPÉRATION DES DONNÉES POST	25
FIGURE 14 : REQUÊTE POST POUR L'ALGORITHME DE RECHERCHE DE CHEF DE PROJET.....	27
FIGURE 15 : CROQUIS DU BROUILLON DE REVUE	29
FIGURE 16 : EXTRAIT DU CODE D'ENREGISTREMENT DE REVUE	30
FIGURE 17 : AFFICHAGE DU BOUTON D'ÉCRASEMENT	30
FIGURE 18 : EXTRAIT DU CODE DE LA CRÉATION DES GRAPHES DE LA REVUE.....	32
FIGURE 19 : EXEMPLE DE DICTIONNAIRE AVEC DONNÉES DE SPRINT	33
FIGURE 20 : EXTRAIT DU CODE POUR AJOUTER LES TÂCHES SELON LEUR ÉTAT.....	34
FIGURE 21 : REQUÊTE POUR OBTENIR TOUTES LES TÂCHES FINIES D'UN PROJET.....	35
FIGURE 22 : EXTRAIT DU CODE POUR RÉCUPÉRER LES PROJETS PARENTS ET LEURS TÂCHES.....	36
FIGURE 23 : DICTIONNAIRE CONTENANT LES PROJETS ET LES INFORMATIONS CONCERNANT CEUX-CI	37
FIGURE 24 : EXEMPLE DE LA MISE EN FORME DES DONNÉES	38
FIGURE 25 : AFFICHAGE DES PROJETS RÉALISÉS PAR UNE ÉQUIPE DURANT UN SPRINT.....	38
FIGURE 26 : EXTRAIT DU CODE DES REQUÊTES À L'API SLACK EN JAVASCRIPT	39
FIGURE 27 : EXTRAIT DU CODE DES REQUÊTES À L'API SLACK EN PYTHON	40
FIGURE 28 : EXEMPLE DU MESSAGE D'ANNONCE DE LA REVUE.....	41
FIGURE 29 : ARCHITECTURE DU PROJET	49
FIGURE 30 : EXEMPLE DES DICTIONNAIRES DE STATISTIQUES.....	50

Table des annexes

ANNEXE 1 : ORGANIGRAMMES DE L'ENTREPRISE.....	II
ANNEXE 2 : OPEN-SPACE ET BUREAU PERSONNEL.....	III
ANNEXE 3 : APERÇU DU MODULE DE SPRINT	IV
ANNEXE 4 : MODÈLE ENTITÉ-ASSOCIATION.....	V
ANNEXE 5 : CAHIER DES CHARGES DU PROJET SCRUM COMPANION	VI
ANNEXE 6 : CROQUIS DE CONCEPTION DE L'APPLICATION « PROJECT WRITING ».....	IX
ANNEXE 7 : MAQUETTE FIGMA DE L'APPLICATION	X
ANNEXE 8 : ENTRÉES ET SORTIES DE L'API JIRA	XI
ANNEXE 9 : CYCLE DE VIE D'UN PROJET JIRA.....	XII
ANNEXE 10 : RÉSULTATS DE L'ALGORITHME DE RECHERCHE.....	XIII
ANNEXE 11 : PAGE DE L'APPLICATION PROJECT WRITING	XIII
ANNEXE 12 : CROQUIS DE CONCEPTION DE L'AFFICHAGE DES PROJETS	XIV
ANNEXE 13 : CAHIER DES CHARGES DU PROJET CAMPUSPLAYLIST	XV
ANNEXE 14 : MAQUETTE DU CAMPUSPLAYLIST	XVII
ANNEXE 15 : MODELE ENTITE-ASSOCIATION DE LA BASE DE DONNEES DU CAMPUSPLAYLIST	XVIII

Glossaire

Les termes définis dans ce glossaire sont identifiables au moyen d'un astérisque (*) lors de la première occurrence.

Atlassian : plateforme d'outils de gestion de projet.

Bitbucket : hébergeur de projets comme Gitlab ou Github, compris dans la suite Atlassian.

Bootstrap : librairie frontend-CSS.

CORS : Cross-Origin Resource Sharing est une norme HTTP gérant les accès aux ressources situées sur une origine (domaine, port, etc.) distincte de celle actuellement utilisée (ex : un domaine-A qui souhaite accéder à une image d'un domaine-B).

Django : Framework de développement web backend Python.

Embed : composant permettant de mettre du contenu interactif sur une page web.

Jira : outil de gestion de projet Agile, compris dans la suite Atlassian.

jQuery : librairie JavaScript.

MVT : patron de conception Model-View-Template.

OKR : sigle « objectifs et résultats clés », méthode de gestion de projet se basant sur des résultats précis à obtenir obligatoirement pour valider les objectifs mis en place.

Référent technique : le référent technique d'un projet a pour responsabilité d'écrire les stories et de répondre aux questions lors du développement. Celui-ci ne peut pas être un développeur.

Slack : logiciel de communication professionnelle.

Ticket : un ticket représente une charge de travail, celui-ci peut être de plusieurs types :

- Epic, représente un rassemblement de plusieurs tickets.
- Task, représente une tâche.
- Story, représente un besoin utilisateur ou technique.
- Bug, représente un bug.
- Sub-Task, représente une sous-tâche d'un ticket.

Token : un token Slack est une chaîne de caractère sous la forme « xob-token », où token représente différents caractères, et permet d'utiliser l'application (bot) pour différentes implémentations, grâce à un système d'authentification.

UNIX : famille de système d'exploitation, comprenant notamment Linux, iOS, macOS, etc.

Introduction

La gestion et la cohésion d'équipe sont des éléments essentiels dans toute entreprise désirant créer un produit de qualité dans les meilleurs délais. L'analyse et l'amélioration des méthodes de travail, la favorisation des échanges ou encore, l'instauration d'un cadre de travail sain et respectueux, pour tous les collaborateurs, sont des aspects et valeurs importants pour GoodBarber.

Dans cette optique, l'entreprise a pris des initiatives pour renforcer ces aspects au sein de ses équipes. Le Scrum Master, en particulier, a lancé deux initiatives clés : la création d'une application de gestion de projet, appelée le « Scrum Companion », et la mise en place d'un jeu musical, nommé le « CampusPlaylist », visant à créer des liens entre les membres de l'entreprise.

Après avoir présenté le contexte de mon stage et notamment, l'entreprise GoodBarber, en expliquant son rôle et son organisation, je présenterai les différents projets et missions qui m'ont été donnés durant ce stage.

Pour chaque projet, j'analyserai l'architecture de celui-ci, si elle existe, ainsi que les technologies mises en œuvre. Par la suite, je présenterai les démarches mises en place pour les différentes implémentations effectuées ainsi que le résultat de celles-ci.

Enfin, je présenterai mes méthodes et mon organisation du travail en entreprise, avant de conclure sur ce que m'a apporté ce stage.

1 Contexte du stage

Dans cette première partie de rapport, je vais expliquer le contexte de mon stage, en présentant notamment l'entreprise et l'environnement dans lequel je travaille, ainsi que les besoins émis et les missions qui m'ont été données.

1.1 Contexte

À l'issue de ma seconde année de BUT Informatique à l'IUT de Montpellier-Sète, j'ai effectué un stage de douze semaines, du 17 avril au 7 juillet 2023 dans le domaine du développement d'application et du développement web.

Ce rapport ne portera que sur les neuf premières semaines du stage.

1.2 Présentation de l'entreprise

L'entreprise qui m'a accueillie pour ce stage est l'entreprise GoodBarber^[1].

Cette SAS (société par actions simplifiées) propose un système de création d'applications « No-Code », c'est-à-dire que les utilisateurs n'ont pas besoin de compétences en développement pour créer leur application et utilisent une interface intuitive proposant différents composants à associer pour concevoir celle-ci.

Leurs clients sont essentiellement des créateurs de contenu, des gérants de boutiques, et des revendeurs. Le business modèle de GoodBarber est la vente d'abonnement^[2] permettant d'avoir les applications à jour.

GoodBarber est implantée mondialement et possède des bureaux à New York, au Portugal, ou encore, en Corse, à Ajaccio où se situe le siège social et se déroule mon stage.

Celle-ci compte quarante-huit employés (quarante à Ajaccio, six au Portugal et deux à New York), répartis dans quatre équipes, telles que : l'équipe Marketing, Customer Success (support client et proposition de services supplémentaires en cas de besoins client), Produit & Design et Technique. L'équipe Technique, quant à elle, est divisée en cinq sous-équipes : PWA (Angular), Android, iOS, Backoffice (PHP), qui forment l'ensemble Frontend, et Backend (API et DevOps). Deux organigrammes de l'entreprise sont trouvables sur [Annexe 1 : Organigrammes de l'entreprise](#)

Concernant l'organisation globale, l'entreprise utilise la suite Atlassian^[3]* et la méthode des OKR*. L'entreprise définit des objectifs globaux et pour chaque objectif, des résultats attendus pour le valider (OKR)^[4]. Chaque équipe définit ensuite ses OKR, basés sur ceux de l'entreprise.

Par la suite, chaque équipe a sa propre organisation :

- L'équipe Technique et l'équipe Marketing utilisent la méthode SCRUM, avec des sprints de 2 semaines.
- L'équipe Technique travaille sur un sujet choisi pendant une semaine, et ce, tous les quatre sprints, afin de combler les dettes techniques ou de développer de nouveaux outils. Ils appellent cela la « White Week ».
- L'équipe Customer Success travaille sur le flux de questions entrant, sans méthode précise, mais avec une réunion quotidienne, permettant le transfert d'informations importantes.

Pour ce qui est de la communication, l'entreprise utilise l'application Slack*, avec des salons par équipe, par sujet ou par projets. De plus, les bureaux d'Ajaccio possèdent une grande salle de réunion avec plusieurs écrans où sont affichés les équipes et membres à distance, grâce à une visioconférence Zoom, afin de réunir tout le monde lors d'événements importants (début et fin de sprint, réunions, etc.).

D'autres petites salles de réunion sont aussi disponibles, pour des réunions nécessitant moins de personnes. Ces réunions et discussions se déroulent en anglais, toutefois, la majorité des personnes parle français.

L'établissement possède également une salle de détente, ainsi qu'un grand open-space dans lequel se trouvent les bureaux des différentes équipes ainsi que mon bureau personnel.

Chaque équipe a un « îlot » de plusieurs bureaux. Mon bureau personnel (cf. [Annexe 2 : Open-space et bureau personnel](#)), quant à lui, est relié à l'îlot de l'équipe Design (n'ayant pas d'équipe attribuée, cela n'a pas d'importance) et comporte un MacBook Pro, prêté par l'entreprise.

Je travaille donc avec deux ordinateurs, mon ordinateur personnel, qui me fait office de deuxième écran et contient tous mes logiciels d'analyse (Figma, Looping, etc.) et le MacBook Pro sur lequel je développe le projet qui m'a été confié.

1.3 Besoins et missions

Mon tuteur est le Scrum Master de l'entreprise et mon « client » principal lors de ce stage. Celui-ci m'a fait part de plusieurs besoins qui correspondent aux missions qui m'ont été données par la suite.

1.3.1 Projet « Scrum Companion »

Mon tuteur possède un outil nommé le « Scrum Companion », composé de différents modules (analyses, administration, gestion de sprints, etc.), qui l'aide au quotidien pour gérer les projets. Cet outil a été créé par différents stagiaires de l'entreprise au cours des années et nécessite à présent de nouvelles implémentations. De ce fait, mon tuteur m'a fait part de plusieurs besoins.

Tout d'abord, il souhaiterait que tout le monde ait une vue générale des projets donnés et des personnes travaillant dessus, pour ne pas perdre le fil. De plus, il aimeraient automatiser différents processus, comme par exemple, le choix du référent technique* d'un projet, afin que ce choix soit calculé et donc, plus juste, ou encore, l'envoi d'annonces et de messages automatiques sur leurs discussions Slack. Enfin, il souhaiterait donner plus d'intérêt à la revue de sprint, en effet, celle-ci ne permet que de générer des chiffres et statistiques par rapport à ce qui a été fait, or, ces chiffres n'aident en aucun cas les différents membres lors des réunions et discussions autour de la revue.

Ces besoins ont un lien avec les différentes missions qui m'ont été données.

La mission principale de ce projet est le développement d'un nouveau module de l'outil « Scrum Companion », permettant à la fois de voir qui travaille, actuellement ou antérieurement, sur quel projet et de choisir automatiquement un référent technique de projet. L'idée est, qu'en un clic, il soit possible de trouver la personne la plus apte à être référent technique d'un projet. Cette fonctionnalité éviterait les potentiels débats, permettrait un gain de temps important et garderait un suivi des réalisations et participations de chacun.

Les trois autres missions, qui sont plus considérées comme des missions secondaires dans ce projet, comprennent : une intégration avec l'API Slack, qui permettra l'automatisation des messages et annonces, avec par exemple, un message annonçant la fin d'un sprint ; l'amélioration d'un module déjà existant, proposant une revue de sprint, où le but serait de rendre la revue disponible avant la fin d'un sprint pour que le Scrum Master puisse avoir un aperçu avant la revue finale ; toujours sur ce même module, la refonte de l'UX afin que les informations affichées de la revue de sprint soient plus intéressantes pour les différents membres d'équipe qui ont accès à l'application.

Toutes ces missions seront analysées plus en détails dans la partie « [2.2 Spécifications fonctionnelles et techniques](#) ».

1.3.2 Projet « CampusPlaylist »

Le CampusPlaylist est un jeu réalisé dans l'entreprise où je travaille pour créer des liens entre les différents membres.

Ce jeu interne consiste à créer une playlist où chacun ajoute une musique, par la suite, chaque joueur doit trouver qui a mis quelle musique dans la playlist. À la fin, les joueurs se réunissent et écoutent ensemble la playlist pour voir les résultats finaux, un système de points et de rôles est mis en place pour donner plus de consistance au jeu.

Une partie est lancée, en général, tous les mois, mais celle-ci doit être initiée par quelqu'un et cette personne est mon tuteur. Tout est fait à la main, les collègues remplissent un formulaire (comme un GoogleForm) et mon tuteur crée la playlist à la main. Or, il se peut que parfois, il n'a pas le temps ou oublie de lancer le jeu, donc personne ne joue. De plus, étant donné qu'il est maître du jeu, il ne peut pas jouer lui-même, car il sait qui ajoute quoi à la playlist.

De ce fait, cela fait maintenant quelques années qu'il souhaite créer une application pour le jeu, afin qu'il puisse enfin jouer ou bien, automatiser les parties, sans passer par des logiciels d'automatisation qui peuvent vite revenir chers. Ma mission pour ce deuxième projet est donc de réaliser cette application, en partant de zéro avec pour seul support, un ancien cahier des charges.

Plus d'informations seront données dans la partie « [3.1 Spécifications fonctionnelles et techniques](#) »

2 Projet « Scrum Companion »

Dans cette partie, je vais présenter le premier projet qui m'a été assigné : l'amélioration de l'outil « Scrum Companion ».

Tout d'abord, je vais décrire les différentes analyses réalisées concernant l'application, les technologies abordées, ainsi que le cahier des charges qui m'a été rédigé. Ensuite, j'aborderai le rapport technique des différentes solutions envisagées et réalisations, ainsi que des éventuels problèmes rencontrés. Enfin, je présenterai un manuel d'utilisation pour l'application, ou du moins, pour les fonctionnalités implémentées.

2.1 Analyse de l'environnement technique

L'outil « Scrum Companion » est l'application que je suis amenée à modifier pour ce premier projet. Celle-ci a été réalisée avec le Framework Django* ainsi que des librairies telles que JQuery* et Bootstrap*.

2.1.1 Problèmes initiaux

Mes premières difficultés, avant même de commencer le projet, étaient que je n'avais jamais utilisé les différentes technologies mises en œuvre. De ce fait, je ne pouvais pas comprendre comment l'application fonctionnait et l'analyse de celle-ci paraissait compliquée, c'est pourquoi j'ai commencé par me former sur Django¹.

Pour ce faire, j'ai suivi les différents tutoriels présents sur le site de Django^[5] mais également sur le site Open Classroom^[6]. C'est grâce à cela que j'ai pu apprendre les bases du Framework. Pour élargir ces bases, j'ai également regardé diverses vidéos sur la chaîne Coding Entrepreneurs^[7], afin d'avoir des démonstrations et des mises en situation plus poussées.

J'expliquerai plus en détails ce que j'ai appris et donc, comment fonctionne Django, dans la partie « [2.1.2 Analyse des technologies](#) ».

¹ Je ne parlerai ici que de ma formation Django, n'ayant pas fait de réelle formation pour Bootstrap et JQuery. La compréhension de ces librairies s'est réalisée plutôt naturellement une fois dans le projet.

Une fois ma formation terminée, ou du moins, assez poussée pour commencer le projet, mon nouvel objectif était de lancer celui-ci et ce ne fut pas aussi simple que cela. Au début, je souhaitais travailler entièrement sur ma machine personnelle, n'étant pas habituée aux ordinateurs Apple. Néanmoins, le projet ne voulait pas se lancer sur Windows, du fait qu'une des fonctions utilisées était spécifique aux systèmes d'exploitation UNIX*.

Après de longues heures de blocage, à chercher une solution, j'ai simplement décidé de régler ce problème en passant sur le MacBook de l'entreprise.

2.1.2 Analyse des technologies

Django est un Framework écrit en Python^[8]. Au début d'un projet, celui-ci est initié par le Framework et contient des modules, aussi appelés « applications », créés par l'utilisateur (cf. [Figure 1](#)). Ces modules sont en réalité des packages contenant des dossiers et fichiers prédéfinis.

Le projet suit un modèle MVT*, contenant les différentes couches suivantes :

- La couche « Model » : un modèle est une classe Python définissant une table de base de données, différents paramètres peuvent être appliqués (comme de l'héritage, de l'abstraction, des contraintes, etc.). Une fois la mise en place des classes terminée, l'utilisateur peut réaliser des « migrations ». Ces migrations correspondent à des mises à jour de la base de données et chaque script de mise à jour est enregistré dans un dossier « migrations », afin de garder une trace des migrations réalisées.
- La couche « View » : les vues sont le cœur de l'application. Un fichier « view.py » contient plusieurs vues, définies par des fonctions ou classes (appelées Class-based views) Python. Celles-ci sont responsables de la gestion des requêtes HTTP et permettent à Django de récupérer les informations à transmettre aux gabarits.
- La couche « Template » : les gabarits sont les fichiers HTML de l'application. Ceux-ci sont très similaires (voire identiques) aux fichiers Twig que nous avions vus en cours. Les gabarits utilisent les données envoyées par les vues et, s'ils existent, des fichiers « statics », qui sont en fait des fichiers CSS et JavaScript, pour générer les pages.

Django possède également une interface d'administrateur, qui permet d'accéder à la base de données, pour par exemple, modifier des données plus facilement (sans passer par des scripts SQL).

Bootstrap est une librairie frontend (CSS) et JQuery une librairie JavaScript.

L'utilisation de Bootstrap est réalisée grâce à l'attribut « class » d'une balise HTML. Afin de charger un style, il suffit de mettre la classe correspondante, trouvable sur les différentes documentations en ligne. Pour JQuery, son utilisation se fait dans un fichier JavaScript.

Dans mon cas, il m'a été demandé d'utiliser les modèles d'Elite Admin^[9]. Ce site répertorie plusieurs composants avec différents styles et il suffit de récupérer l'architecture HTML et les classes correspondantes pour les appliquer au projet.

2.1.3 Architecture du projet

L'architecture de l'application est gérée automatiquement par Django.

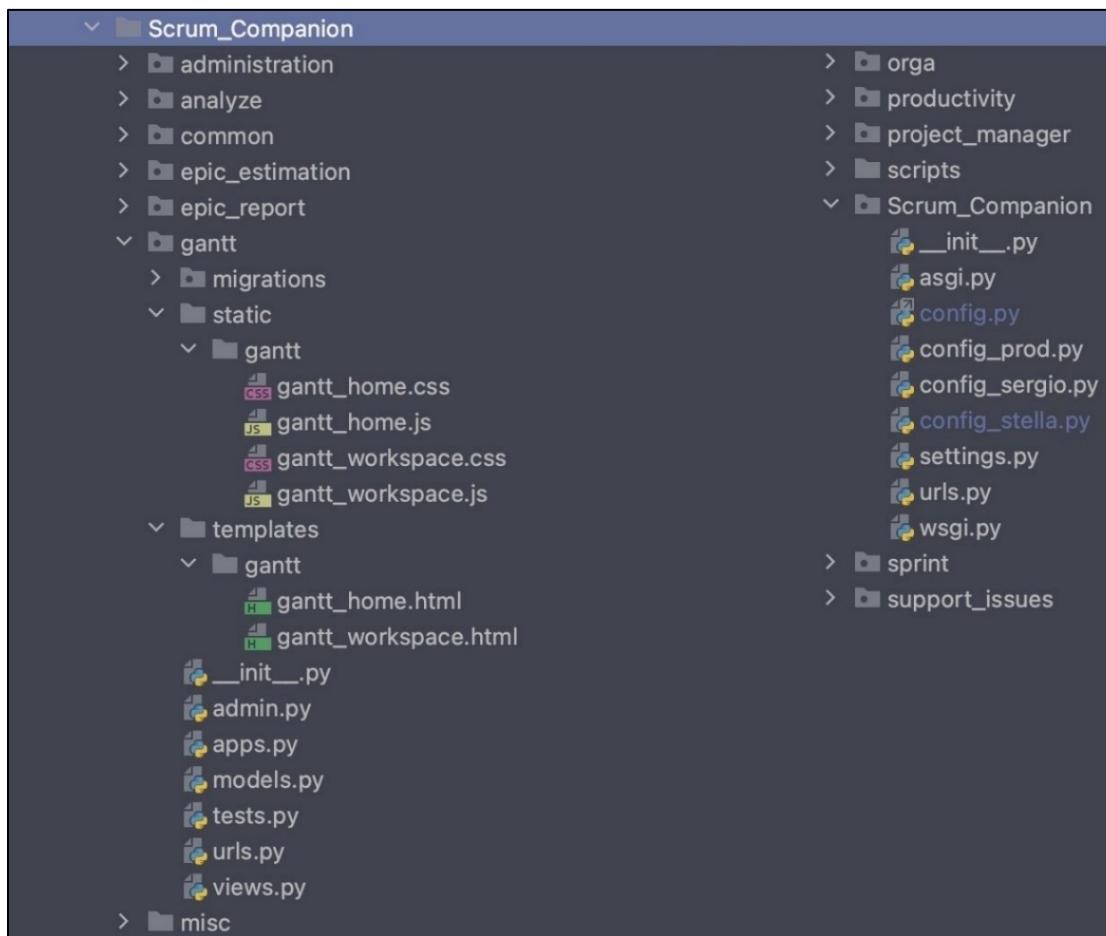


Figure 1 : architecture du projet

Dans un projet (ici, « Scrum Companion »), tous les dossiers et fichiers prédéfinis des modules sont identiques, à l'exception du module « Scrum Companion », qui porte le même nom que le projet et contient les fichiers de mise en place de l'application.

Dans un module « normal », nous retrouvons donc (cf. [Figure 1](#), module « gantt ») :

- Les dossiers suivants :
 - o « migrations », contenant l'historique des changements des modèles dans la base de données.
 - o « static », contenant tous les fichiers CSS, JavaScript ou encore, les assets.
 - o « templates », contenant les fichiers HTML.
- Les fichiers suivants :
 - o « `_init.py_` », initiant le module.
 - o « `admin.py` », contenant les informations liées au module à enregistrer dans le gestionnaire d'administrateur.
 - o « `apps.py` », contenant les informations à propos du module.
 - o « `models.py` », contenant les modèles de base de données.
 - o « `tests.py` », contenant les tests unitaires.
 - o « `urls.py` », contenant les liens précis du module.
 - o « `views.py` », contenant les vues.

Tous ces fichiers sont liés au module auquel ils appartiennent et il peut, bien évidemment, y avoir des fichiers ajoutés manuellement.

Concernant le module « Scrum Companion », celui-ci contient (cf. [Figure 1](#), module « Scrum Companion ») :

- Les fichiers suivants :
 - o « `_init.py_` », initiant le module.
 - o « `asgi.py` », contenant la mise en place de l'interface ASGI².
 - o « `config.py` », contenant la configuration de la base de données, généré grâce à un lien symbolique au fichier `config_nom.py`.
 - o « `settings.py` », contenant toutes les configurations du projet, avec par exemple, une liste de tous les modules à utiliser, l'host du site, la connexion à la base de données, etc.
 - o « `urls.py` », contenant les liens principaux du site, se situant généralement avant un lien de module précis, par exemple : « /sprint/analyse » ou « /sprint/détail », où sprint est le lien principal et la suite, le lien précis.
 - o « `wsgi.py` », contenant la mise en place de l'interface WSGI³.

² <https://asgi.readthedocs.io/en/latest/>

³ <https://wsgi.readthedocs.io/en/latest/>

Par ailleurs, voici un schéma montrant l'architecture autour du projet (cf. [Figure 2](#)) :

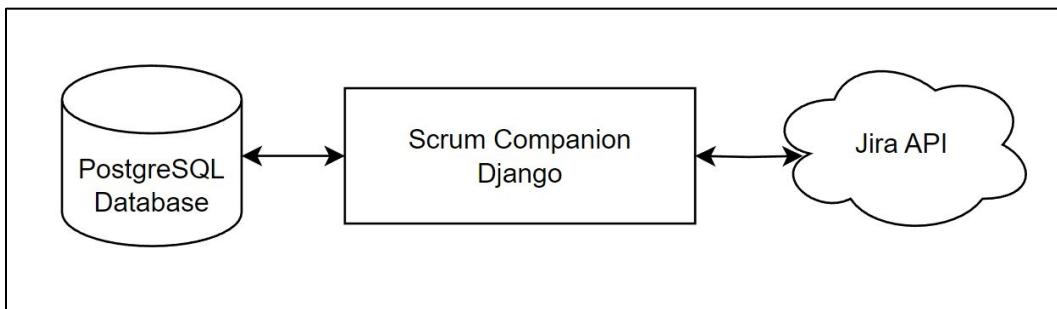


Figure 2 : architecture externe du projet

Celui-ci utilise une base de données PostgreSQL pour stocker les données relatives à l'application. De plus, une connexion à l'API Jira* est effectuée pour obtenir les informations concernant les sprints, les projets, etc.

2.1.4 Architecture de la base de données

Pour mieux comprendre les différentes données que j'avais à disposition lors de ce projet, j'ai réalisé le modèle entité-association de la base de données, que vous trouverez sur [Annexe 4 : Modèle Entité-Association](#).

Dans mon cas, seules certaines tables de la partie « orga.models » et la table « ReviewState » me seront utiles :

- La table « Person », qui est une table abstraite, pourrait permettre l'héritage d'une nouvelle table, contenant les personnes qui ne sont pas des développeurs, ou bien, être transformée en table normale, afin qu'une personne puisse être instanciée.
- La table « Developer » permettra de récupérer le nom et prénom, l'ID Jira et l'état d'un développeur. L'état sert à conserver une trace des développeurs ayant quitté l'entreprise (pour des statistiques par exemple), nous définissons alors un état « INACTIVE ».
- La table « Platform » servira à obtenir les différentes sous-équipes de l'équipe Technique et de faire la liaison entre un développeur et son équipe affectée.
- La table « ReviewState » concerne la revue d'un sprint et pourra être modifiée pour la mission de « Preview de Review ».

Par ailleurs, je possède ma propre base de données PostgreSQL en local pour stocker mes données et réaliser mes tests.

2.1.5 Application et fonctionnalités

Le Scrum Companion comporte différents modules (cf. [Figure 3](#)).



Figure 3 : liste des fonctionnalités du Scrum Companion

La première application, « Sprint » (cf. [Annexe 3 : Aperçu du module de Sprint](#)) contient toutes les informations concernant un sprint actuel ou futur. Nous pouvons y retrouver les différents objectifs du sprint, la revue, la rétrospective ou encore, un compteur déterminant la fin du sprint. Ces sprints peuvent être administrés (supprimés ou modifiés) sur la page « Administration ».

Les applications « Epic Estimation » et « Epic Report » comportent différentes informations concernant les projets (nommés « Epics »). Nous pouvons y retrouver des statistiques concernant le nombre de tâches réalisées ou à réaliser par équipe sur un projet, par exemple.

L'application « Gantt » permet la réalisation d'un diagramme de Gantt et l'application « Support Issues » affiche des statistiques concernant les bugs de l'application, sur un intervalle temporel.

Enfin, les applications « Productivity » et « Analyze » permettent d'avoir des statistiques sur la productivité et sur les réalisations lors d'un sprint. L'application « Analyze » contient aussi une liste de projets présents dans le Backlog.

Lors de mon projet, je serai amenée à modifier l'application « Sprint » et plus précisément, la gestion de la revue de sprint. Je devrai également ajouter une nouvelle application pour la gestion de l'écriture d'un projet.

2.2 Spécifications fonctionnelles et techniques

Vous retrouverez, sur [Annexe 5 : Cahier des charges du projet Scrum Companion](#), le cahier des charges rédigé par mon tuteur concernant le projet du « Scrum Companion ».

Ma mission principale, celle d'ajouter un nouveau module à l'application, est donc divisée en 3 parties : l'affichage de la charge courante des développeurs ; l'affichage d'un historique des projets et la création d'un algorithme de recherche.

Voici donc, mes interprétations pour ce cahier des charges : les différents affichages seront très similaires, seules les données changeront, la charge courante se ferait sur la période actuelle et l'historique sur les 365 derniers jours.

La liste demandée serait sûrement un tableau, avec trois colonnes : « nom », « nombre de projets » et « liste des projets ». Le tri demandé se ferait alors sur la deuxième colonne, par ordre décroissant. Les projets contenus dans la troisième colonne pourraient être filtrés par phase, avec le nombre de projets renseigné pour chaque phase, et les développeurs par équipes. Il faudrait aussi afficher le nombre de tâches pour chaque projet à côté de celui-ci.

Les données pour remplir ce tableau seraient récupérées dans les tables « Developer » et « Platform » vues précédemment. Il restera à trouver un moyen pour récupérer les projets d'une personne en particulier ainsi que les informations les concernant.

Concernant l'algorithme, celui-ci reprendra les informations du tableau de la charge courante. Nous voulons qu'il renvoie la personne ayant le moins de charge actuellement et qui est membre de l'une des équipes d'une liste donnée.

Pour les bonus, il faudrait voir comment est gérée la criticité d'un projet, et par exemple, ajouter un poids à un projet (comme une priorité en Scrum) et recalculer le résultat de l'algorithme en fonction de celui-ci. L'algorithme retournerait alors la personne ayant le moins de charge actuellement et, si deux personnes ont la même charge, nous souhaitons la personne avec le moins de poids de criticité.

Concernant l'affichage des personnes n'étant pas des développeurs, nous pourrions utiliser l'état « INACTIVE » du modèle de base de données, pour les personnes n'étant plus là ou encore, effectuer des modifications sur le model « Person », comme émis précédemment.

Pour ce qui est des missions secondaires, nous en comptons trois : l'amélioration de la revue de sprint, l'ajout d'une prévisualisation de la revue et une intégration à l'application Slack.

La liste des projets, demandée dans l'amélioration de la revue, pourrait reprendre la liste des projets de l'historique, avec ici, non pas les 365 derniers jours, mais les quatorze derniers jours (durée d'un sprint). Il faudrait réaliser plusieurs maquettes afin d'avoir une idée plus claire que celle présentée dans le cahier des charges.

Concernant la prévisualisation de revue, il faudrait éviter l'enregistrement des données de la revue actuelle. Cela pourrait se faire en créant une nouvelle table, contenant une ou plusieurs revues temporaires et en empêchant la revue actuelle d'être enregistrée avant la fin du sprint. De ce fait, tant que la revue officielle n'est pas enregistrée, nous afficherions les informations de la dernière revue temporaire réalisée.

Enfin, pour l'intégration Slack, cela pourrait être réalisé grâce à un bot Slack. Ce bot serait lié aux événements du site et enverrait les messages lorsqu'un événement a lieu.

2.3 Rapport technique

Dans cette partie, je vais présenter les différentes réalisations lors de ce projet, avec notamment, les démarches réalisées, les problèmes et solutions trouvées ainsi que les résultats finaux.

2.3.1 Application « Project Writing »

Avant de commencer la réalisation de l'application, il fallait imaginer le fonctionnement de celle-ci ainsi que son visuel. De ce fait, plusieurs brouillons pour l'application ont été réalisés et vous retrouverez sur [*Annexe 6 : Croquis de conception de l'application « Project Writing »*](#), le croquis final de l'application.

La page comprend trois sections déroulantes :

La première section concerne la charge courante, représentée par un tableau contenant l'équipe du développeur (sous forme de pastille colorée), son nom, son nombre de projets actuel ainsi que la liste de ces mêmes projets.

Le tableau est trié par nombre de projets descendant et la liste des projets dans la troisième colonne par nombre de points d'efforts. Différents filtres sont aussi disponibles afin de n'afficher que les lignes avec les membres de certaines équipes ou encore, de n'afficher que les projets dans certains états.

La seconde section est l'historique, très semblable à la charge courante, la seule différence étant les filtres. Un projet dans l'historique n'a pas plusieurs états, celui-ci ne peut être que terminé. De ce fait, pour remplacer celui-ci, un filtre par année pourrait être implémenté, pour avoir par exemple, les projets de 2022, 2021, etc.

Un système de pagination sur les tableaux est envisagé, dans le cas où il y aurait trop de données à afficher, afin de ne pas surcharger la page. De plus, lors d'un clic sur un des projets de la troisième colonne, deux options seraient possibles : l'affichage d'une carte, contenant des informations sur le projet ou alors, une simple redirection vers sa page Jira.

La troisième, et dernière section de la page, est l'algorithme de recherche de chef de projet. Cet algorithme prendrait deux paramètres : le premier désignant la forme du résultat voulu (soit une personne unique, soit une liste de personnes) et le second étant la liste des équipes où nous voulons chercher les membres.

Le résultat obtenu montre alors un ou plusieurs développeurs, avec l'équipe, le nom et la charge actuelle de celui-ci, ainsi qu'un bouton « Contacter » afin de prévenir la personne qu'elle a été choisie, en utilisant la future intégration Slack.

La majorité de la solution a été validée par le client, par ailleurs les idées de cartes pour les projets et de liste pour les résultats de l'algorithme n'ont pas été retenues. Par conséquent, les options alternatives de ces implémentations ont été choisies : une redirection Jira pour les projets et une seule personne pour le résultat de l'algorithme avec possibilité de relancer en excluant cette personne. De plus, celui-ci souhaitait que la charge courante et la recherche de chef de projet soient séparés de l'historique et que les non-développeurs ainsi que le bouton « Contacter » n'apparaissent plus.

Afin de mieux visualiser la future implémentation, une maquette Figma pour la page principale a été créée (cf. [Annexe 7 : Maquette Figma de l'application](#)), la page d'historique reprendra grandement cet affichage. Cette initiative met d'ailleurs en relation mon apprentissage en développement d'interfaces utilisateur (apprentissage critique 1.1) et représente le point de départ du gabarit Django.

Pour la première implémentation de l'application, la priorité était l'affichage de la charge courante, étant donné que celle de l'historique était grandement identique mais moins complète.

La page est créée à partir d'une simple vue et envoie, au gabarit lié, les développeurs présents dans la base de données. Par la suite, le tableau est généré sur le gabarit grâce à des éléments Bootstrap ainsi qu'une boucle « for » qui boucle sur les développeurs. Chaque ligne du tableau est créée à partir des données du développeur bouclé, de ce fait, voici le code réalisé (cf. [Figure 4](#)).



```

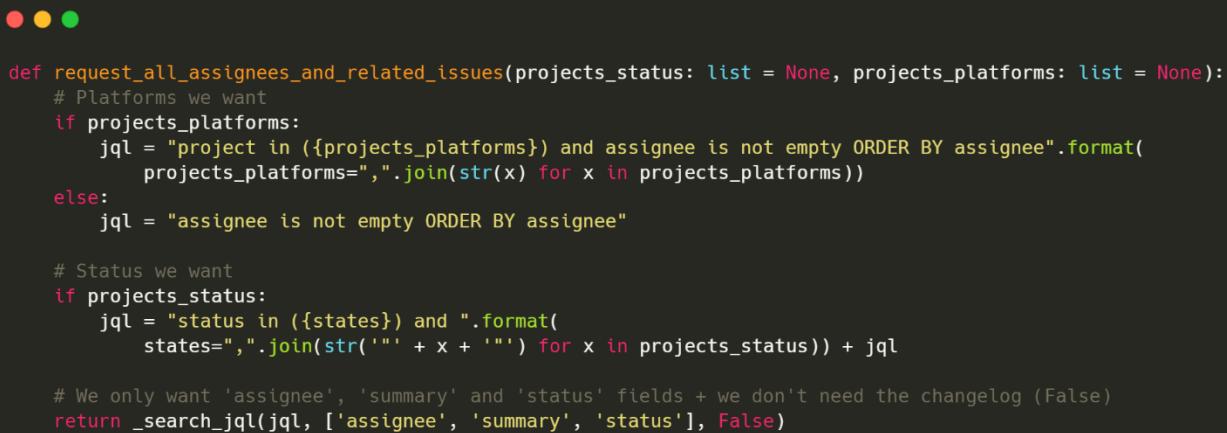
<table class="table primary-bordered-table color-bordered-table m-b-0">
    <thead>
        <tr>
            <th>Developers</th>
            <th>Nb projects</th>
            <th>Current projects</th>
        </tr>
    </thead>
    <tbody>
        <!--list of devs-->
        {% for developer in developers %}
            <tr>
                <td>
                    <i class="iconeCouleurEquipe"></i> {{nomDeveloppeur}}
                </td>
                <td>futurNombreProjets</td>
                <td>futureListeProjets</td>
            </tr>
        {% endfor %}
    </tbody>
</table>

```

Figure 4 : exemple de code pour le tableau de charge courante

À présent, il faut trouver un moyen de remplir les colonnes « futurNombreProjets » et « futureListeProjets » de chaque développeur. Ces données ne sont pas enregistrées sur la base de données, il est donc obligatoire de réaliser une requête à l'API Jira.

Jira utilise un système de recherche permettant d'obtenir des tickets* spécifiques. Le système de recherche avancé fonctionne de la même façon qu'une requête de base de données, par exemple, pour obtenir les tickets de type « bugs », ayant pour statut « à faire », la requête est la suivante : « issuetype = Bug AND status = "To Do" », cela renvoie alors une liste comprenant tous les résultats. De ce fait, une nouvelle requête Jira a été formulée (cf. [Figure 5](#)) :



```

def request_all_assignees_and_related_issues(projects_status: list = None, projects_platforms: list = None):
    # Platforms we want
    if projects_platforms:
        jql = "project in ({}) and assignee is not empty ORDER BY assignee".format(
            projects_platforms=", ".join(str(x) for x in projects_platforms))
    else:
        jql = "assignee is not empty ORDER BY assignee"

    # Status we want
    if projects_status:
        jql = "status in ({}) and {}".format(
            projects_status, "status=" + ", ".join(str('"' + x + '"') for x in projects_status)) + jql

    # We only want 'assignee', 'summary' and 'status' fields + we don't need the changelog (False)
    return _search_jql(jql, ['assignee', 'summary', 'status'], False)

```

Figure 5 : requête à l'API Jira pour avoir les personnes assignées aux projets

Cette requête prend pour paramètres une liste de statuts voulus et une liste d'équipes voulues puis renvoie un JSON contenant les tickets résultants de la requête « jq1 ».

L'appel à la méthode « _search_jql » va effectuer l'appel à l'API avec la requête « jq1 », le tableau en second argument permet de restreindre les résultats à certains champs du JSON, ici, nous ne voulons que les personnes assignées aux tickets, le nom du ticket (son résumé) et son statut.

Dans notre cas, la liste de statuts voulus contient les statuts « à faire », « écriture en cours » et « développement en cours », utilisés pour les filtres, quant à la liste des équipes, celle-ci est vide car l'affichage nécessite toutes les équipes. Un schéma expliquant les entrées et sorties de l'API Jira est disponible sur [Annexe 8 : Entrées et sorties de l'API Jira](#).

Une fois la requête effectuée, les résultats sont envoyés au gabarit et, pour afficher les projets d'une personne dans le tableau, nous vérifions si la personne assignée est bien la personne stockée en base de données, si c'est le cas, nous lui ajoutons le projet.

Pour ce qui est du nombre de projets, celui-ci est le nombre d'éléments présents dans la troisième colonne du tableau, ce choix a été majoritairement réalisé pour les filtres par états. Si nous sélectionnons les projets « à faire », le nombre de projets affiché doit correspondre au nombre de projets « à faire » du développeur, ce qui n'est pas possible si nous comptons tous les projets du développeur dans le résultat de l'API. Par ailleurs, le tri de cette colonne a d'ailleurs été implémenté en reprenant le code présent sur un tutoriel W3S^[10].

Voici à quoi ressemble l'affichage du tableau actuellement (cf. [Figure 6](#)) :

Developers	Nb issues	Current issues
Mathieu Fancello	16	 iOS-3244 (3) iOS-3220 (2) iOS-3193 (2) CM-112 (2) CM-108 (2) CM-94 (2) CM-83 (2) CM-64 (2) CM-52 (2) CM-48 (2) CM-46 (2) CM-13 (2) iOS-3198 (1) FRONTEND-440 (0) FRONTEND-5 (0) BACKEND-407 (0)
Laurent Linza	11	 FRONTEND-849 (0) FRONTEND-423 (0) FRONTEND-405 (0) FRONTEND-376 (0) FRONTEND-276 (0) FRONTEND-6 (0) BACKEND-462 (0) BACKEND-388 (0) BACKEND-287 (0) BACKEND-195 (0) BACKEND-170 (0)
Jim Marchetti-Ettori	6	 FRONTEND-14549 (6) WEB-1542 (3) FRONTEND-14550 (2) FRONTEND-9 (0) EB-760 (0) BACKEND-348 (0)
Marc Leonardi	2	 BACKEND-3015 (0) BACKEND-2948 (0)
Pierre- Laurent Medori	2	 BACKEND-3365 (0) BACKEND-3257 (0)

Figure 6 : première version du tableau de charge courante

Chaque développeur à son équipe (sous forme d'une pastille colorée) et ses « projets », colorés selon leurs états, avec une icône présentant le type de projet et le nombre de points d'efforts associé.

Cependant, les données affichées ne sont pas celles voulues par le client. En effet, les projets sont en réalité des tâches de projets, or, le client voulait le projet en lui-même et non pas ses enfants. De plus, actuellement, tous les développeurs sont affichés, mais si un développeur ne travaille plus pour l'entreprise (donc son statut dans la base de données est « INACTIVE »), il ne doit pas apparaître.

Pour corriger cela, il est nécessaire de filtrer les développeurs selon leur état et il faut modifier la requête Jira pour n'inclure que les tickets de type « Epic » (qui sont les projets), présents dans les tableaux Jira « EB » et « IB ». Des explications plus détaillées du contexte de la gestion des projets Jira sont disponibles sur [Annexe 9 : Cycle de vie d'un projet Jira](#).

Nous obtenons donc la requête suivante (cf. [Figure 7](#)) :



```
jql = "assignee not in inactiveUsers() and project in ({projects}) and \
statusCategory != Done and type = EPIC and \
assignee is not empty ORDER BY assignee"
```

Figure 7 : requête Jira pour obtenir les résultats de la charge courante

Celle-ci peut être traduite par la récupération des projets (Epic), non finis (!=Done) présents dans les tableaux EB et IB ({projects} qui est formaté par la suite) où l'assigné existe et n'est pas inactif.

Cette implémentation permet d'obtenir toutes les données nécessaires pour l'affichage sauf les points d'efforts du projet. En effet, ceux-ci sont calculés à partir des points d'efforts des tâches du projet, de ce fait, il faut, pour chaque projet, effectuer une requête Jira pour obtenir toutes les tâches qui lui sont associées et ensuite, additionner les points d'efforts de ces tâches.

Une requête Jira existe déjà pour retrouver les enfants d'un projet, mais celle-ci prend en paramètre une liste de clés de projets pour réduire le temps de requête. De ce fait, l'idée est

la suivante : lors de la mise en forme pour l'envoi des données au gabarit, toutes les clés des projets et la liste de toutes les tâches seront récupérées au même moment.

Par la suite, il faut assigner de nouveau un projet à une tâche, en vérifiant que le parent de cette tâche est bien le projet, si c'est le cas, un compteur est incrémenté avec le nombre de points d'efforts de la tâche. Faire ceci pour toutes les tâches permet d'obtenir l'addition des points d'efforts et donc, le nombre de points total de chaque projet.

Voici un visuel de la solution (cf. [Figure 8](#)) :

```
● ● ●

projects = [] # list of all projects information
projects_ids = [] # list of all projects ids

for project_JSON in JSON:
    # we fetch the values from the JSON for each project,
    # and we store them in a dict, that will be stored in a list
    project = {.....}
    projects.append(project)

    projects_ids.append(project_JSON.id) # we put the id in the ids list

# we request all the stories, based on the list of projects ids (if it's not empty)
stories = []
if projects_ids:
    stories = request_todo_stories_in_epics(projects_ids)

# for each dict (project) in our list (projects)
for project in projects:
    for story in stories:
        # if the project is the parent of the story, we add the points
        if project.get('project_id') == story.parent.id:
            project['story_points'] += story.storypoints
```

Figure 8 : code récupérant les points d'efforts d'un projet

La première partie de celle-ci concerne la mise en forme des données. Pour tous les projets présents dans le JSON renvoyé par la requête Jira, seules les données intéressantes sont récupérées et stockées dans un dictionnaire nommé « projet ». Ce dictionnaire est, par la suite, stocké dans une liste « projects » qui contiendra tous les projets (sous forme de dictionnaires, avec leurs données à l'intérieur).

À présent, voici (cf. [Figure 9](#)) à quoi ressemble le tableau. Nous pouvons voir que le nombre de « projets » a grandement réduit, ce qui est plus cohérent.

Persons	Nb projects ▾	Current projects
● Lisa Colombani	5	EB-660 2.5 EB-659 2.5 IB-12075 0 EB-766 0 EB-563 0
● Jim Marchetti-Ettori	2	EB-760 18.5 EB-764 18
● Flo Luccioni	2	EB-763 21 EB-747 13

Figure 9 : version finale du tableau de charge courante

Étant donné que la page d'historique est identique à celle de la charge courante, un héritage de gabarits Django a été mis en place pour éviter la redondance. De ce fait, il y a un fichier HTML contenant les éléments communs aux deux pages, un fichier HTML transmettant les affichages spécialisés de la charge courante et un fichier HTML transmettant ceux de l'historique. Ces affichages spécifiques sont gérés grâce aux blocks Django, ceux-ci sont similaires à ceux que nous avions vus en cours avec Twig et permettent d'afficher des données différentes selon la page qui est affichée ainsi que deux vues, l'une chargeant la page HTML de la charge courante et l'autre, de l'historique.

Les données de l'historique sont différentes de celles de la charge courante. En effet, pour l'historique, nous souhaitons avoir les projets terminés sur les 365 derniers jours et nous voulons aussi afficher les développeurs inactifs. De ce fait, la requête Jira a été modifiée pour réaliser deux requêtes différentes, pour les deux cas de demandes (charge courante ou historique).

Voici la nouvelle requête Jira (cf. [Figure 10](#)). Cette requête prend deux nouveaux arguments, deux booléens « history » et « active » et renvoie les résultats de la requête Jira selon ces arguments.

Dans le cas où nous aurions « history=False » et « active=True », nous demandons les résultats pour la charge courante, comme expliqué plus haut. À l'inverse, nous demandons les résultats de l'historique, c'est-à-dire, les projets (Epic), finis (!=Done) et publiés (published) durant les 365 derniers jours (PublicationDate), présents dans le tableau GTA, qui est le tableau des projets finis, où l'assigné existe, peu importe son état (actif ou inactif).

```

● ● ●

def request_projects_and_assignees(history: bool = False, active: bool = True, additional_fields: list = None):
    # find every project where there's at least an assignee
    jql = "type = EPIC and assignee is not empty ORDER BY assignee"

    # find projects done during the last 365 days in GTA
    if history:
        jql = "project = {project} and 'Publication Date[Date]' >= -365d and statusCategory = Done and status =
              {published} and ".format(
            project=utils.PROJECT_GTA.key,
            published=utils.STATUS_PUBLISHED.id) + jql
    # find current projects in EB, IB
    else:
        jql = "project in ({projects}) and statusCategory != Done and ".format(
            projects=".join(str(x) for x in [utils.PROJECT_EB.key, utils.PROJECT_IB.key])) + jql

    # find only active assignees
    if active:
        jql = "assignee not in inactiveUsers() and " + jql

    fields = ['assignee', 'issuetype', 'project', 'status', 'summary']
    # add additional fields to the query
    if additional_fields:
        fields.extend(additional_fields)

    return _search_jql(jql, fields, False)

```

Figure 10 : requête finale pour les données des tableaux

La méthode utilisée pour récupérer les points d'efforts d'un projet prend beaucoup de temps à être réalisée, surtout lors du chargement de l'historique où des projets avec plus de 200 points sont présents. C'est pourquoi, afin de ne pas relancer cette méthode à chaque chargement de page, les résultats à afficher sont stockés dans une session Django.

Ce stockage en session est réinitialisé tous les jours pour l'historique et à chaque chargement de page pour la charge courante, si jamais des modifications ont eu lieu dans la journée, et permet un chargement plus rapide, voire instantané de la page. Cette initiative me permet notamment de montrer que je sais déjà anticiper des résultats tels que des temps d'exécution, ce qui est un apprentissage prévu pour la troisième année de BUT Informatique (apprentissage critique 2.3).

Maintenant que l'implémentation des tableaux est effectuée, passons à la gestion des filtres. Il y a cinq filtres pour les équipes et trois filtres pour les projets, l'historique compte un sixième filtre pour les équipes pour le cas où la personne est inactive. Leur fonctionnement est le suivant : lors d'un clic sur un bouton de filtre, les éléments correspondants sont masqués. Par exemple, tous les filtres sont activés au départ, en cliquant sur le bouton « IOS », tous les membres faisant partie de l'équipe IOS disparaissent et le bouton IOS se désactive, en pressant de nouveau dessus, celui-ci se réactivera et les membres seront de nouveau affichés.

Le changement est réalisé côté JavaScript. Pour tous les boutons de filtres, un écouteur va appeler la fonction « appliquerFiltre » sur le bouton pressé. Pour un filtre d'équipe, nous nous référons à la pastille du développeur, qui, dans ses classes contient l'équipe (pour afficher la couleur), pour un filtre d'état de projet, nous nous référons à l'élément HTML du projet en lui-même, qui possède pour classe son état.

Le problème est que pour cacher un projet, il suffit de lui mettre un affichage « hidden » mais, en faisant la même chose sur la pastille, cela ne va pas cacher la ligne entière. C'est pourquoi, lors de l'appel à « appliquerFiltre », une vérification a lieu pour savoir quelle est la source de l'appel, si c'est un projet, dans ce cas, l'élément du projet est masqué, sinon, le grand-parent de la pastille, qui correspond à la ligne du tableau, est masqué.

Pour cacher et afficher les éléments, la propriété CSS « display » est utilisée ainsi que des classes indiquant le style du bouton de filtrage. Pour tous les éléments concernés par le filtre, si l'élément est déjà caché, alors, il est affiché et le bouton du filtre est activé, sinon, l'élément est caché et son bouton, désactivé (au visuel, un bouton activé est un bouton rempli et un bouton désactivé est un bouton vide).

Vous retrouverez un visuel du fonctionnement sur ce lien : [Démo-filtres](#)

Lors de la validation de cette première implémentation par le client, celui-ci souhaitait retirer le filtre « écriture en cours », qui n'était pas important pour l'affichage, et il n'était surtout pas d'accord avec le fonctionnement implémenté. En général, lors de l'utilisation d'un filtre, celui-ci affiche tous les éléments le concernant, dans notre cas, c'est l'inverse, en cliquant dessus cela retire tous les éléments le concernant.

Pour régler ce problème, le code a entièrement été revu. L'utilisation de la propriété HTML « aria-pressed », qui, lorsqu'utilisée sur un bouton Bootstrap, permet d'ajouter une classe « active » ou de la retirer selon la valeur du booléen de la propriété, était le changement majoritaire. De ce fait, pour gérer le style des boutons actifs ou inactifs, plutôt que d'ajouter et de retirer des classes aux boutons, il suffit à présent de les gérer grâce aux sélecteurs CSS et de faire en sorte que, si le bouton contient la classe active, le style du bouton actif est appliqué.

Par ailleurs, cela permet aussi d'obtenir une liste des boutons actifs. Cette information sert à gérer les cas où tous les boutons sont inactifs ou tous les boutons sont actifs.

Dans le cas où tous les boutons sont actifs (la liste est complète), cela signifie qu'il n'y a pas besoin de filtrer, donc les filtres doivent être désactivés, en réinitialisant l'affichage et en forçant la propriété « aria-pressed » à « false ». Ce même mécanisme se déroule quand tous les boutons sont inactifs, là aussi, il n'y a pas besoin de filtrer donc une réinitialisation à lieu.

Pour ce qui est du filtrage dans les autres cas, tous les éléments liés aux boutons inactifs sont cachés et tous les éléments liés aux boutons actifs sont affichés. De ce fait, lors d'un clic sur le bouton « IOS », tous les développeurs qui ne sont pas de l'équipe IOS doivent disparaître (leurs boutons sont inactifs) et ceux qui sont de l'équipe IOS doivent rester (leur bouton est actif).

Voici le code JavaScript expliqué plus tôt (cf. [Figure 11](#)) :

```
● ● ●

function applyPlatformFilters() {
    // we store all current active buttons
    const activePlatformsButtons = document.querySelectorAll("div.platforms-filters button.active")

    // if there's no or full actives, we reset the display
    if (activePlatformsButtons.length === 0 || activePlatformsButtons.length === platformsFiltersButtons.length) {
        listAllPlatforms()
    }
    // otherwise, we just want the regular mechanism
    else {
        // we remove the display of every platform
        for (let button of platformsFiltersButtons) {
            changePlatformsDisplays(button, 'none')
        }
        // we activate the display of active platforms only
        for (let button of activePlatformsButtons) {
            changePlatformsDisplays(button, 'table-row')
        }
    }
}
```

Figure 11 : extrait du code pour filtrer les équipes

Ce code est spécifique aux équipes et est lié à un écouteur sur les boutons. Les filtres pour les états de projets ont un code similaire où certains éléments changent. Il a été décidé de séparer les deux cas en plusieurs fonctions pour plus de clarté dans le code.

Vous retrouverez un visuel du fonctionnement sur ce lien : [Démo-filtres-finaux](#)

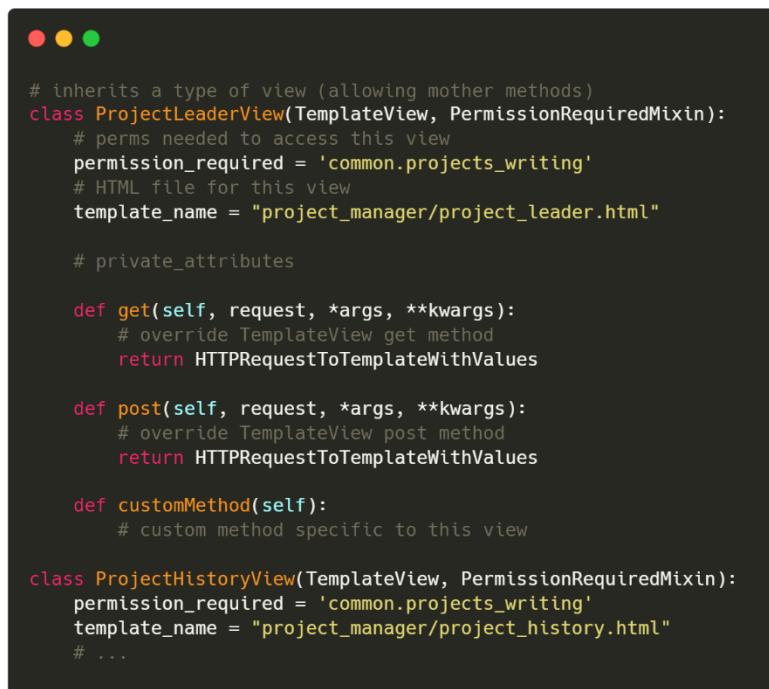
Enfin, concernant cette application, il ne reste qu'à expliquer l'implémentation de la recherche de chef de projet.

L'idée pour cet algorithme était de récupérer, grâce au sélecteur de « platforms », en bas à gauche de [Annexe 7 : Maquette Figma de l'application](#), une liste d'équipes, envoyée dans un

formulaire POST. Selon le bouton cliqué (roll, re-roll ou reset request), nous aurions eu des comportements différents à gérer du côté de la vue.

Dans le cas d'un premier « roll », l'algorithme cherche une personne qui a pour équipe l'une des équipes de la liste envoyée par le formulaire. En cas de « re-roll », l'algorithme est relancé avec la même liste d'équipe mais la personne trouvée précédemment est exclue des personnes potentielles. Enfin, en cas de « reset request » ou d'un second « roll », les listes des équipes et des personnes exclues sont vidées ou modifiées.

Pour effectuer ceci, il fallait modifier la structure actuelle des vues qui n'étaient que des fonctions Python renvoyant une requête HTTP. Celles-ci ont donc été remplacées par des « class-based views » (cf. [Figure 12](#)), ces types de vues permettent de définir des éléments appartenant à celle-ci et d'obtenir des méthodes héritées de classes déjà créées par Django.



```
# inherits a type of view (allowing mother methods)
class ProjectLeaderView(TemplateView, PermissionRequiredMixin):
    # perms needed to access this view
    permission_required = 'common.projects_writing'
    # HTML file for this view
    template_name = "project_manager/project_leader.html"

    # private_attributes

    def get(self, request, *args, **kwargs):
        # override TemplateView get method
        return HttpResponseRedirectWithValues

    def post(self, request, *args, **kwargs):
        # override TemplateView post method
        return HttpResponseRedirectWithValues

    def customMethod(self):
        # custom method specific to this view

class ProjectHistoryView(TemplateView, PermissionRequiredMixin):
    permission_required = 'common.projects_writing'
    template_name = "project_manager/project_history.html"
    # ...
```

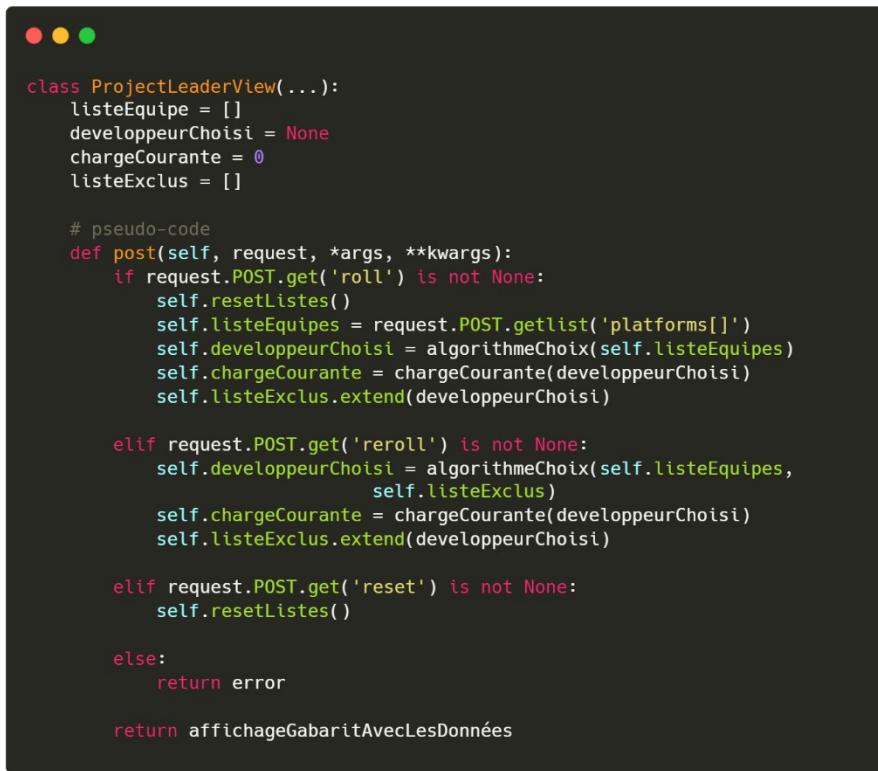
Figure 12 : exemple de class-based views

Il est par la suite possible de remplacer ces méthodes pour changer leur fonctionnement, dans notre cas, nous souhaitons modifier le fonctionnement de la méthode POST et ajouter d'autres méthodes.

Du côté HTML, le formulaire comprend dans son architecture le sélecteur ainsi les trois boutons, chaque bouton a un nom, de ce fait, lors de l'envoi du formulaire en méthode POST, il est possible de récupérer la liste des équipes et le bouton cliqué dans les arguments de la requête. Par ailleurs, j'ai décidé d'utiliser une librairie nommée « Select2 »[\[11\]](#) pour gérer le

sélecteur d'équipe, celle-ci permettait d'avoir exactement l'aspect souhaité pour celui-ci, je l'ai donc rajouté au projet et mis en place.

Par la suite, du côté de la vue Python, les données sont récupérées grâce à la requête POST et l'algorithme est lancé selon le type de bouton pressé (cf. [Figure 13](#)) :



```

class ProjectLeaderView(...):
    listeEquipe = []
    developpeurChoisi = None
    chargeCourante = 0
    listeExclus = []

    # pseudo-code
    def post(self, request, *args, **kwargs):
        if request.POST.get('roll') is not None:
            self.resetListes()
            self.listeEquipes = request.POST.getlist('platforms[]')
            self.developpeurChoisi = algorithmeChoix(self.listeEquipes)
            self.chargeCourante = chargeCourante(developpeurChoisi)
            self.listeExclus.extend(developpeurChoisi)

        elif request.POST.get('reroll') is not None:
            self.developpeurChoisi = algorithmeChoix(self.listeEquipes,
                                                       self.listeExclus)
            self.chargeCourante = chargeCourante(developpeurChoisi)
            self.listeExclus.extend(developpeurChoisi)

        elif request.POST.get('reset') is not None:
            self.resetListes()

        else:
            return error

    return affichageGabaritAvecLesDonnées

```

Figure 13 : première version de la récupération des données POST

Différents attributs sont stockés dans la vue, ceux-ci pourront être réutilisés par la suite, en cas de nouvelle requête POST, ces attributs sont : la liste des équipes précédemment récupérée, le développeur choisi lors du dernier algorithme, sa charge courante et une liste des développeurs exclus pour les prochains algorithmes.

Lors d'un « roll », si elles existent, les listes des équipes et des exclus sont réinitialisées grâce à la méthode « resetListes() », cela incite à utiliser « re-roll » si nous souhaitons relancer plusieurs fois l'algorithme. Par la suite, la liste des équipes présentes dans le sélecteur est récupérée grâce à la requête POST, cette liste permet d'obtenir le développeur courant et la charge courante en lançant l'algorithme de choix (celui-ci sera décrit plus tard) et le développeur choisi est ajouté à la liste des exclus.

Cette liste des exclus est utilisée en cas de « re-roll », dans ce cas, nous souhaitons obtenir un nouveau développeur, différent des précédents trouvés. Lors d'un « reset », nous allons

tout simplement réinitialisé, comme lors d'un roll. Par ailleurs, si aucun des trois boutons n'est cliqué, mais qu'une requête POST est quand même envoyée, alors il y a une erreur.

Enfin, nous retournons l'affichage de la page, avec les nouvelles données (le développeur et sa charge, stockés dans la vue) et le résultat de l'algorithme (cf. [Annexe 10 : Résultats de l'algorithme de recherche](#)).

Concernant l'algorithme de choix en lui-même, une liste des développeurs est créée, ceux-ci sont triés par nombre de projets en cours croissants (retrouvable grâce aux données récupérées précédemment pour l'affichage de la charge courante et de l'historique), de ce fait, en parcourant cette liste, les premiers développeurs auront zéro projet, puis un projet, etc.

Lors du parcours, une vérification a lieu pour savoir si le développeur fait partie d'une des équipes de la liste voulue, si c'est le cas alors, il est renvoyé dans deux cas : s'il n'y a pas de liste d'exclus ou s'il n'en fait pas partie. S'il ne permet pas ces conditions, l'algorithme passe au prochain développeur et lorsqu'aucun développeur n'est disponible, celui-ci renvoie « Not Found » qui permettra d'afficher le bouton « reset request » dans le gabarit Django.

Cette implémentation a de nombreux problèmes que le client a fait remonter.

Tout d'abord, le sélecteur utilisé n'est pas le meilleur en termes d'UX, en effet, si l'utilisateur veut choisir les cinq équipes, il doit cliquer dix fois sur celui-ci (une fois pour ouvrir le sélecteur et une fois pour sélectionner l'équipe), un système de boutons activables serait plus ergonomique.

Ensuite, lorsque nous lançons l'algorithme, nous obligeons la page à se recharger à chaque fois à cause de l'envoi du formulaire et, lorsque nous souhaitons recharger la page dans pour autant lancer l'algorithme, un pop-up indiquant que la requête POST sera renvoyée s'affiche.

Enfin, l'algorithme de choix en lui-même est très basique et pourrait être amélioré, en rajoutant par exemple, un poids en fonction de l'état d'un projet, pour départager les développeurs en cas d'égalité.

Le sélecteur a donc été modifié et la librairie « Select2 » retirée afin de mettre cinq boutons. Par ailleurs, la récupération des équipes est maintenant réalisée en JavaScript plutôt qu'en Python, étant donné que les boutons ont un état « actif » et « inactif », nous récupérons seulement les valeurs des boutons « actifs » pour former la liste des équipes.

À présent, pour éviter le chargement de la page à chaque lancement de l'algorithme, une requête POST en JavaScript a été créée. Cette requête est la suivante (cf. [Figure 14](#)) :



```

function searchAlgorithmData(boutonDeclenché) {
    // body data
    let body = {
        "platforms[]": listeDesPlateformes // (boutons actifs),
        "type": boutonDeclenché // (roll, reroll ou reset request)
    }

    let request = createRequest("vuePython", 'POST', body);

    // call to the algorithm
    fetch(request)
        .then(function (response) {
            if (!response.ok) {
                throw Error(response.statusText);
            }
            return response.json();
        })
        // results of the algorithm
        .then(function (json_result) {
            if (type === 'roll' || type === 'reroll') {
                showLuckyWheel(true); // ajout d'un loader personnalisé type roue de la fortune
                setTimeout(() => {
                    showLuckyWheel(false)
                })
                if (type === 'roll') {
                    // changes div visibility only when we roll (= once)
                }
            }

            const person = json_result.person

            if (!($.isEmptyObject(person))) { // si quelqu'un a été trouvé par l'algorithme
                // display changes based on the new developer found
                showToast("success", "Someone has been found ! :)");
            } else {
                // displays the not found card
                showToast("error", "No one has been found :(");
            }
        })
        .catch(function (error) {
            showToast("error", error);
        });
}

```

Figure 14 : requête POST pour l'algorithme de recherche de chef de projet

Le type de bouton enclenché et la liste des plateformes, récupérée grâce aux boutons « actifs », sont envoyés dans le corps de la requête.

Par la suite, dans la vue, l'algorithme est effectué selon le bouton enclenché, de la même façon qu'avant. Cependant, l'algorithme prend maintenant une liste triée par nombre de projets et par facteur de projet croissants, ce facteur est calculé pour chaque développeur au chargement de la page et stocké dans les données de celle-ci (cf. [page 21](#)), de ce fait, l'algorithme est quasiment instantané, car il n'a pas besoin de calculer de nouveau les données.

Le facteur de projet représente la somme des points d'efforts de tous ses projets, selon leur état (voir [Annexe 9 : Cycle de vie d'un projet Jira](#)). Par exemple, un projet A en cours d'écriture et un projet B à faire, le projet A demande plus de ressources que le projet B, de ce fait, son poids équivaut à ses points d'efforts multipliés par 1.5, le poids du projet B quant à lui, reste le nombre de points d'efforts normal. La somme de tous ces poids de projets permet d'avoir un facteur de projet, lorsque deux développeurs ont la même charge de projet (par exemple, deux projets chacun), nous allons regarder ce facteur et prendre celui qui a le facteur le plus petit.

Les résultats de l'algorithme sont à présent envoyés sous forme de réponse JSON, avec les plateformes précédemment choisies et la personne choisie (dictionnaire contenant ses informations et son nombre de projets actuel). De ce fait, le JavaScript va attendre les données, grâce à l'indication « fetch » qui attend une promesse et ne recharge pas la page, si la promesse n'échoue pas, dans ce cas, nous allons gérer les affichages des éléments HTML avec les données reçues.

Dans le cas où nous faisons un « roll » ou un « re-roll », une icône de chargement est lancée, celle-ci représente une roue de la Fortune, demandée par le client, je l'ai donc créée en utilisant un logiciel de dessin et implémentée en JavaScript. Un cas spécifique est codé pour le « roll » afin d'afficher des éléments qui ne doivent pas être affichés lors d'un « re-roll », sinon, nous vérifions qu'un développeur a été trouvé par l'algorithme, si c'est le cas, nous affichons la carte des résultats, sinon, nous affichons la carte d'un résultat vide. Dans le cas d'un « reset request », nous réinitialisons tous les affichages mis en place précédemment et nous désactivons les boutons d'équipes.

L'aspect final de l'application « Project Writing » est retrouvable sur [Annexe 11 : Page de l'application Project Writing](#).

2.3.2 Preview de Review

L'idée de base de cette fonctionnalité, comme énoncée dans le cahier des charges, était de créer une revue de sprint sans pour autant l'enregistrer dans la base de données. De ce fait, la première conception était la suivante (cf. [Figure 15](#)) :

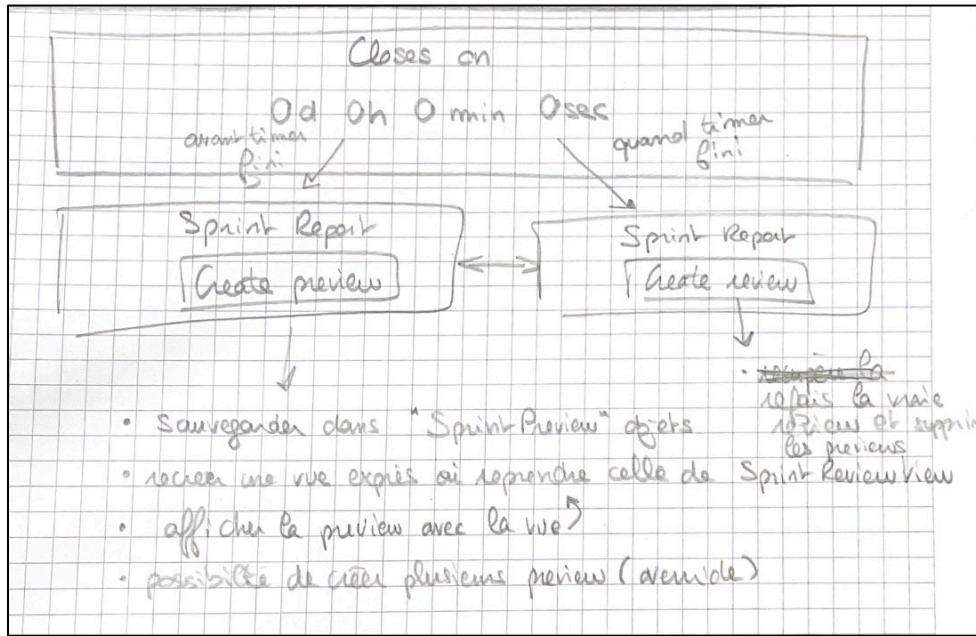


Figure 15 : croquis du brouillon de revue

En se basant sur le temps restant du sprint, il y aurait eu plusieurs interfaces :

La première interface, lorsque le sprint est toujours en cours, afficherait un bouton « Créer un brouillon de revue ». À la pression de ce bouton, la revue aurait été sauvegardée dans la base de données, dans un nouveau modèle (table) appelé « SprintPreview ».

Par la suite, le Scrum Master pourrait cliquer sur un bouton « Voir brouillon » ou « Voir revue » qui proposerait un choix parmi une liste de brouillons ou alors, dans le cas où nous n'en voudrions qu'un seul, qui afficherait le brouillon associé au sprint actuel.

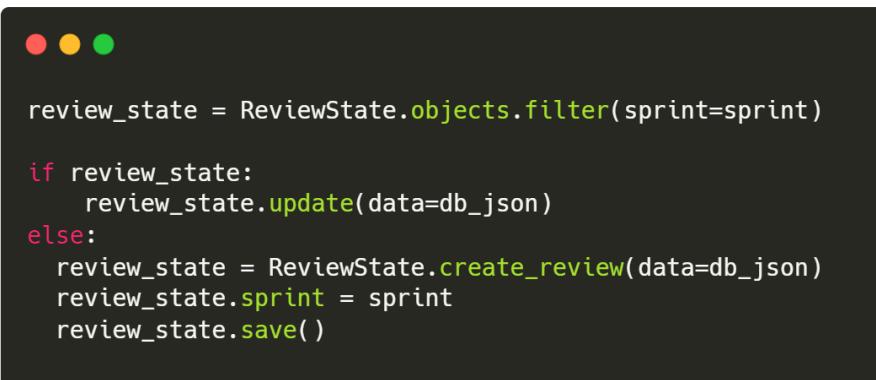
La seconde interface, lorsque le sprint est fini, afficherait cette fois-ci un bouton « Créer la revue » qui aurait récupéré le dernier brouillon sauvegardé et l'aurait enregistré dans la table « ReviewState », en effaçant les autres brouillons.

Une fois la revue sauvegardée, il y aurait eu l'interface normale avec un bouton « Voir revue ».

Cette solution est loin d'être la meilleure, elle demande de modifier la structure de la base de données seulement pour enregistrer des objets temporaires et pourrait perdre les utilisateurs avec les différentes interfaces et brouillons.

La seconde conception est beaucoup plus simple en termes de compréhension et d'usage, elle est notamment celle qui a été validée par le client. Le but de celle-ci est d'écraser la revue existante par une nouvelle revue.

Dans le code (cf. [Figure 16](#)), la revue associée au sprint est récupérée grâce à la table « ReviewState ». Dans le cas où une revue existerait déjà, une mise à jour des données est effectuée (grâce à la méthode update, en passant à la clé « data », la valeur « db_json ») au contraire, si aucune revue n'existe pour ce sprint, une revue est créée avec le code fourni de base.



```
review_state = ReviewState.objects.filter(sprint=sprint)

if review_state:
    review_state.update(data=db_json)
else:
    review_state = ReviewState.create_review(data=db_json)
    review_state.sprint = sprint
    review_state.save()
```

Figure 16 : extrait du code d'enregistrement de revue

De ce fait, lorsque nous arrivons sur la page, un bouton « Créer revue » est présent et, une fois activé, laisse apparaître deux boutons, « Voir revue » et « Écraser revue » (cf. [Figure 17](#)).

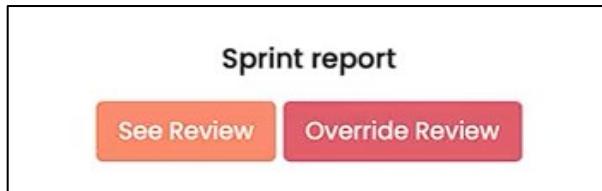


Figure 17 : affichage du bouton d'écrasement

2.3.3 Amélioration de la Review

Avant de commencer les améliorations de la revue, il fallait obtenir plus d'informations sur ce qui était voulu. Grâce au cahier des charges, je savais que le client souhaitait afficher les projets travaillés durant le sprint, pour autant, il était de mon ressort de choisir l'UX de cet affichage, c'est pourquoi différentes solutions ont été proposées (cf. [Annexe 12 : Croquis de conception de l'affichage des projets](#)).

La première idée était d'afficher les projets avec différentes informations concernant ceux-ci, par exemple, les tâches du projet, un diagramme montrant les avancées des stories, etc. Ces projets auraient été accessibles depuis une liste de boutons à leurs noms, afin de ne pas afficher toutes les informations de tous les projets directement sur la page.

Finalement, le nom, l'état et le diagramme du projet ont été retenus pour l'affichage. Ce diagramme comprend : les tâches réalisées par l'équipe avant ce sprint, celles réalisées durant ce sprint, celles à réaliser et celles en cours de réalisation. De plus, lorsqu'un projet est un objectif du sprint, il a été décidé de mettre un signe distinctif pour le montrer.

Lors de l'implémentation de la solution choisie, j'ai remarqué que la page HTML de la revue ne respectait pas du tout le principe DRY (Don't Repeat Yourself), de ce fait, je ne savais pas vraiment comment m'y prendre et comment implémenter au mieux mon code.

La page comporte différentes sections (blocs de balises HTML), chacune est répétée cinq fois afin d'afficher les données des cinq équipes, seules ces données changent et la structure HTML reste la même, ce qui représente un fichier de 2500 lignes répétitives. Sans changement là-dessus, l'affichage des projets aurait dû être ajouté cinq fois, pour les cinq équipes différentes, c'est pourquoi j'ai pris l'initiative de refaire la page, en instaurant le plus possible le principe DRY.

Afin de remédier à ceci, une liste d'index est définie dans la vue de la revue, celle-ci est ensuite envoyée au gabarit. Ces index sont tout simplement les cinq équipes de l'entreprise, par la suite, il suffirait de modifier cette liste pour ajouter ou supprimer une équipe de l'affichage. Cette liste d'index pourrait aussi être automatisée, mais cela demanderait des modifications plus poussées que le client ne souhaite pas.

Voici un exemple de l'implémentation, avec un code JavaScript initiant les graphes de la revue (cf. [Figure 18](#)) :

```
{% for index in indexes %}
    {% with platform_data=index|lower|add:'_data' %}

        /*** STORIES ***/
        new Chart(
            document.getElementById('{{index|lower}}_stories_chart'),
            {
                type: 'doughnut',
                data: {
                    labels: [
                        'To Do',
                        'In Progress',
                        'Conformity Review',
                        'Done'
                    ],
                    datasets: [{
                        label: 'Number of issues',
                        data: [
                            {% with issues=sprint_data|keyvalue:platform_data|keyvalue:'issues'%}
                                '{{ issues.2|length}}',
                                '{{ issues.4|length}}',
                                '{{ issues.10631|length}}',
                                '{{ issues.3|length}}'
                            {% endwith %}
                        ],
                        backgroundColor: [
                            chart_todo_color,
                            chart_inprogress_color,
                            chart_conformity_color,
                            chart_done_color
                        ],
                        hoverOffset: 4
                    }]
                },
            }
        )
    }

    /*** AUTRES GRAPHES ***/
    ...
    {% endwith %}
{% endfor %}
```

Figure 18 : extrait du code de la création des graphes de la revue

Dans le gabarit, une boucle sur les index définis précédemment est effectuée, de ce fait, il est possible d'obtenir les noms des équipes bouclées et il sera possible d'afficher les données correspondantes à celles-ci.

Pour récupérer les données d'une équipe, il faut récupérer une clé de dictionnaire appelée « nom-équipe_data », pour ce faire, une variable est redéfinie à chaque boucle et contiendra le nom de l'équipe suivi de « _data », grâce au tag Django « with » (ex : si l'index est backend, la variable « platform_data » sera « backend_data » et nous pourrons ensuite récupérer les données backend avec cette variable).

Par la suite, nous souhaitons créer un graphe avec les données de la plateforme actuellement bouclée. Ce graphe nécessite un « id » pour être affiché, dans notre cas, cet « id » correspond au nom de l'équipe suivi du nom du graphe (ex : « backend_stories_chart ») et possède une balise retrouvable dans le code, là où sera affiché le projet.

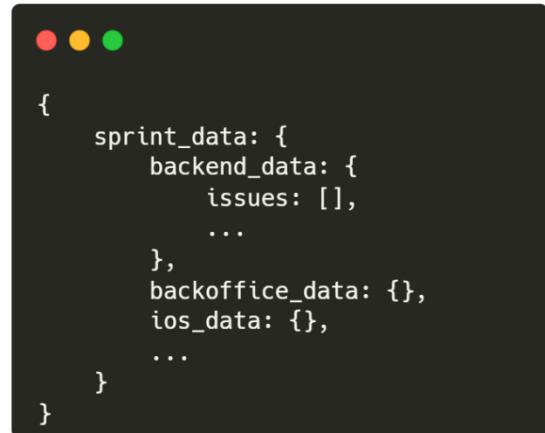
Enfin, il suffit de récupérer les données à mettre dans la section « data » du graphe. Ces données sont récupérées grâce à la variable « platform_data » définie plus haut, chaque donnée d'équipe est stockée dans un dictionnaire, celui-ci est de la forme suivante (cf. [Figure 19](#)).

Pour récupérer la donnée « issues », il faut récupérer le « platform_data » contenu dans le « sprint_data ». De ce fait, le tag « with » est encore une fois utilisé pour créer une variable, celle-ci contiendra la donnée « issues » de la plateforme actuellement bouclée.

Par ailleurs, afin d'obtenir la valeur de la clé, j'ai créé un filtre Django personnalisé « |keyvalue » qui retourne la valeur à l'emplacement de la clé dans le dictionnaire. Cela signifie que la donnée « issues » correspond à : `sprint_data[platform_data][issues]`.

En effectuant ce changement sur tout le code et en remplaçant la redondance par des boucles, celui-ci passe de 2500 lignes à 500 lignes environ. Cela permet maintenant d'implémenter la solution plus facilement et efficacement, tout en mettant en relation mes apprentissages en termes de bonnes pratiques de conception et de programmation (apprentissage critique 1.2).

Par la suite, il fallait encore étudier le code déjà présent dans la vue Python, afin d'ajouter la récupération des données nécessaires pour l'affichage. Le code de cette fonctionnalité comprend deux parties (qui sont des fonctions) : une première partie où nous allons enregistrer la revue dans la base de données, en enregistrant toutes les données relatives au sprint dedans (par exemple, les tâches du sprint), sous forme de dictionnaires ; une deuxième partie qui, lors de l'appel à la page de la revue, va charger ces données et les mettre en forme pour chaque équipe (elle est donc appelée cinq fois).



```
{
    sprint_data: {
        backend_data: {
            issues: [],
            ...
        },
        backoffice_data: {},
        ios_data: {},
        ...
    }
}
```

Figure 19 : exemple de dictionnaire avec données de sprint

La première implémentation de la solution réutilisait grandement le code de la deuxième partie. Celui-ci comporte une boucle avec toutes les tâches réalisées sur le sprint, ces tâches sont des dictionnaires avec différentes informations, dont l'information concernant leur parent, le projet que nous cherchons à afficher. En récupérant le projet, nous pouvons facilement récupérer son nom et son état.

Par la suite, pour obtenir les différentes tâches d'un projet, il suffit de vérifier si la tâche bouclée a un parent et de la stocker si c'est le cas.



```
project_done_issues_by_key = defaultdict(list) # => {key : [...]}
project_todo_issues_by_key = defaultdict(list)

for issue in issues_of_platform:
    ### Original code doing something ###

    if status_category_id == common.utils.STATUS_CATEGORY_DONE.id:
        ### Original code doing something ###

        # Getting the done issues of each project (if the project exists)
        if issue.parent:
            project_done_issues_by_key[issue.parent].append(issue.key)

    # To-do stories
    elif status_category_id == common.utils.STATUS_CATEGORY_TODO.id:
        if issue.parent:
            project_todo_issues_by_key[issue.parent].append(issue.key)

#project_done_issues_by_key => {projectName1: [issue1, issue2], projectName2: [issue5, issue6]}
```

Figure 20 : extrait du code pour ajouter les tâches selon leur état

Dans l'exemple ci-dessus (cf. [Figure 20](#)), nous avons deux dictionnaires, le premier contenant les tâches finies d'un projet et le second les tâches en cours d'un projet.

Lorsque le statut de la tâche bouclée est « Fini », nous l'ajoutons à la liste présente à la clé portant le nom de son parent (par exemple, le parent est EB-XXX, nous aurons { EB-XXX : [tâcheBouclée1, tâcheBouclée5...]} dans le dictionnaire des tâches finies. Sinon, si sa catégorie est « À faire », nous réalisons la même chose, mais avec le dictionnaire des tâches en cours.

Nous avons donc les tâches finies durant ce sprint et celles à faire, pour chaque équipe. Dans cette première implémentation, nous ne souhaitions pas avoir les tâches en cours, il ne manque alors que les tâches réalisées avant ce sprint.

Pour obtenir ces tâches, une requête à l'API Jira sera effectuée. La requête est la suivante (cf. [Figure 21](#)) :

```
● ● ●  
def request_done_issues_by_platform_in_epics(epic_key, platform_name):  
    jql = "\"Epic Link\" = {epic_key} AND statusCategory = Done AND Platform[DropDown] =  
    {platform_name}".format(epic_key=epic_key, platform_name=platform_name)  
  
    return _search_jql(jql)
```

Figure 21 : requête pour obtenir toutes les tâches finies d'un projet

La requête prend comme argument la clé du projet et l'équipe concernée et renvoie un dictionnaire contenant toutes les tâches finies par l'équipe sur le projet. Ce dictionnaire peut ensuite être transformé en liste pour ne garder que la clé de la tâche.

À présent, il suffit d'envoyer au gabarit les différentes données concernant un projet.

Un projet est défini par son nom, possède une catégorie, une liste de tâches terminées durant le sprint, une liste de tâches à faire et une liste de tâches terminées. Le graphe nécessite le nombre de tâches pour chaque cas, il faut donc envoyer la taille des différentes listes qui équivaudra à ce nombre. Pour ce qui est des tâches terminées, celles-ci comprennent aussi les tâches terminées durant le sprint, c'est pourquoi il faut soustraire le nombre de celles-ci pour obtenir le bon résultat.

Plusieurs problèmes apparaissent dans cette première implémentation.

Tout d'abord, les tâches à faire ne concernent que celles qui ont été choisies pour le sprint, or, en dehors du sprint, le projet peut avoir un backlog de tâches à faire.

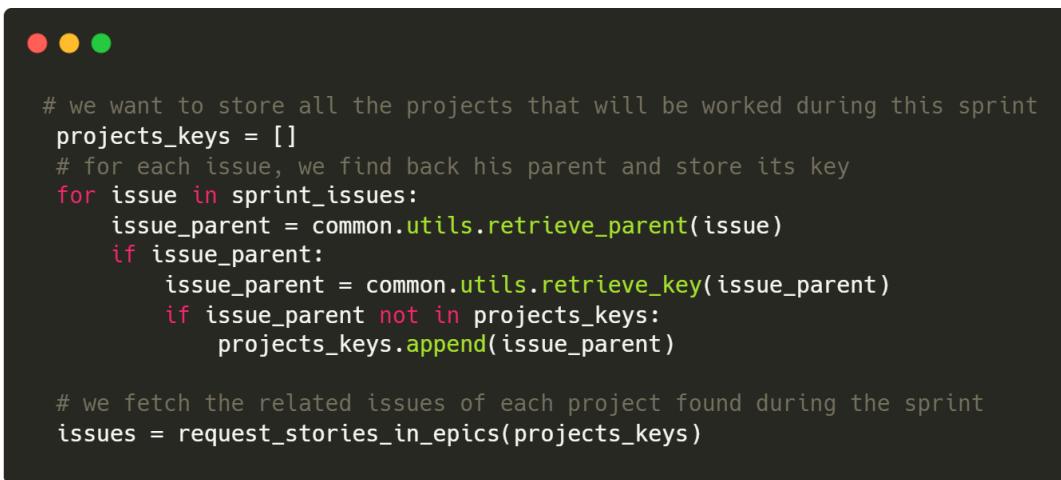
Ensuite, étant donné que cette implémentation se trouve dans la deuxième partie du code, celle-ci est appelée dès que la page de revue est affichée. De ce fait, la requête Jira sera effectuée de nouveau, à chaque chargement de la page et il y a un risque que les données changent entre temps, ce qui n'est pas souhaité.

De plus, la gestion des projets par équipe était erronée. Certains projets ne s'affichaient pas, car ils n'étaient pas pris en compte par l'algorithme du fait que seuls les projets ayant des tâches terminées étaient utilisés.

Enfin, comme dit plus haut, cette implémentation ne prend pas en compte les tâches en cours de réalisation et les objectifs du sprint.

Une deuxième implémentation a donc été réalisée. Plutôt que de récupérer les données dans la deuxième partie du code, celles-ci seront récupérées dans la première partie et donc, enregistrée dans la base de données. De ce fait, elles ne seront plus modifiables ou du moins, que lorsque la revue sera recréée grâce à la fonctionnalité de [Preview de Review](#).

Tout d'abord, il fallait que trouver un nouveau moyen pour récupérer tous les projets travaillés durant le sprint. Pour ce faire, nous avions à notre disposition la liste des tâches des équipes Backend et Frontend, il suffisait de fusionner celles-ci pour avoir toutes les tâches du sprint, puis, de parcourir ces tâches et de stocker les clés de leurs parents (les projets) dans une liste (cf. [Figure 22](#)).



```
# we want to store all the projects that will be worked during this sprint
projects_keys = []
# for each issue, we find back his parent and store its key
for issue in sprint_issues:
    issue_parent = common.utils.retrieve_parent(issue)
    if issue_parent:
        issue_parent = common.utils.retrieve_key(issue_parent)
        if issue_parent not in projects_keys:
            projects_keys.append(issue_parent)

# we fetch the related issues of each project found during the sprint
issues = request_stories_in_epics(projects_keys)
```

Figure 22 : extrait du code pour récupérer les projets parents et leurs tâches

La liste de clés sert à réaliser une requête à l'API Jira. Cette requête prend en paramètre une liste de clés de projets et renvoie toutes les tâches de ces projets. Grâce à ça, il est possible d'avoir accès aux tâches en dehors du sprint, comme par exemple, les tâches à faire.

À présent, nous possédons les tâches de tous les projets dans un même tableau mais nous n'avons pas de liste de tâches associées à un projet (ex : projetA = [tâche1, tâche2, ...]). Seule la tâche possède l'information sur son parent, le parent, lui, ne connaît pas ses enfants.

Pour régler ce problème, nous aurions pu ne pas utiliser une liste de clés pour la requête mais utiliser la clé elle-même, ce qui aurait permis d'associer automatiquement le projet aux tâches trouvées. Le problème avec cette solution est que, pour un nombre x de projets, il y aurait eu un même nombre x de requêtes, et donc, un temps de calcul beaucoup plus long.

Une autre solution a été choisie, celle d'associer de nouveau les données grâce à une boucle imbriquée.

D'abord, les projets sont parcourus, durant ce parcours, les tâches sont également parcourues et nous vérifions si la tâche parcourue appartient au projet. Dans le cas où la tâche appartient au projet, l'état et le nom du projet sont récupérés, puis la tâche est ajoutée à une liste, sinon, nous passons à la prochaine tâche.

Ces informations récupérées sont directement stockées dans un dictionnaire avec pour clé, la clé du projet. À la fin, tous les projets sont stockés dans un dictionnaire « projets », qui est stocké dans la base de données (cf. [Figure 23](#)).



```
'projets' {
    'clé-projet1' : {
        'nom-projet1': nom1,
        'état-projet1': état1,
        'tâches-projet1': [{}], {}
    }
    'clé-projet2' : {
        'nom-projet2': nom2,
        'état-projet2': état2,
        'tâches-projet2': [{}], {}
    }
}
```

Figure 23 : dictionnaire contenant les projets et les informations concernant ceux-ci

Pour ce qui est de la mise en forme de ces nouvelles données, l'implémentation des données finies durant le sprint a été gardée ainsi que la liste des projets réalisés par l'équipe durant le sprint.

Il fallait donc récupérer les données concernant les tâches finies en dehors du sprint ainsi que les tâches à réaliser et les tâches en cours. Pour ce faire, l'idée était de parcourir tous les projets présents dans la base de données et également toutes les tâches de chacun de ces projets (boucle imbriquée).

Si l'équipe ayant la tâche attribuée est l'équipe que souhaitée (celle-ci est passée en paramètre), dans ce cas, les états de la tâche sont vérifiés. Si la tâche est finie, celle-ci est ajoutée à une liste de tâches finies pour ce projet et la même chose est réalisée pour les tâches en cours et les tâches à faire. Cela donne alors trois dictionnaires de la forme : { 'projet1' : [tâche1, tâche2], 'projet2' : [tâche3, tâche4, tâche5], etc... }.

Ensuite, il fallait mettre en forme ces données, pour les envoyer au gabarit et les afficher (cf. [Figure 24](#)). Un dictionnaire, contenant la clé du projet et toutes ses données, est alors stocké dans un dictionnaire nommé « projets ».

Cette action se déroule toujours à l'intérieur du parcours de projet mais en dehors du parcours de tâches. Il est nécessaire de vérifier que le projet parcouru est bien un projet

travaillé durant ce sprint par cette équipe, sinon, il se peut que des projets non voulus apparaissent.

```
# We only want to see current projects
if clé-projet in clé-projets-actuels:
    projets.update(
        {clé-projet: {
            'nom': nomProjet,
            'état': étatProjet,
            'tâches-finies': taille(tableauTâchesFinies[clé-projet]),
            'tâches-finies-avant-sprint': taille(tableauTâchesFiniesAvantSprint[clé-projet]) -
                taille(tableauTâchesFinies[clé-projet]),
            'tâches-à-faire': taille(tableauTâchesAFaire[clé-projet]),
            'tâches-en-cours': taille(tableauTâchesEnCours[clé-projet])
        }
    }
)
```

Figure 24 : exemple de la mise en forme des données

Le dictionnaire « projets » contient donc tous les projets réalisés par l'équipe durant le sprint ainsi que toutes les informations nécessaires concernant celui-ci pour créer les graphes.

La dernière chose à faire du côté de l'algorithme est de distinguer les projets qui sont des objectifs du sprint de ceux qui ne le sont pas. Pour ce faire, le modèle de base de données « Goals » contient un attribut correspondant à la clé du projet, une liste contenant toutes ces clés d'objectifs est donc transmise au gabarit.

Dans le gabarit, tous les projets contenus dans mon dictionnaire créé précédemment sont parcourus et un affichage est généré pour chacun, avec un graphe dédié. Une vérification est également lieu pour savoir si la clé du projet est contenue dans les clés d'objectifs, si c'est le cas, alors une étoile dessus apparaît. Par ailleurs, les projets sont triés dans un ordre précis, ceux étant des objectifs sont prioritaires à ceux qui ne le sont pas.

Voici la réalisation finale après avoir mis en place les données dans le gabarit (cf. [Figure 25](#)) :

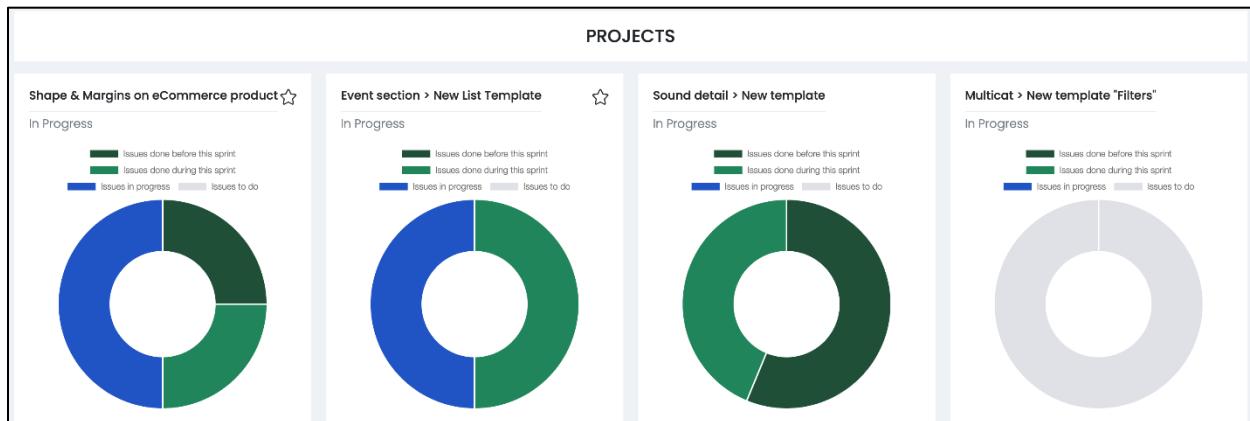


Figure 25 : affichage des projets réalisés par une équipe durant un sprint

2.3.4 Intégration Slack

Enfin, pour cette dernière mission, deux solutions s'ouvraient à moi.

La première était de réaliser une intégration Slack avec la librairie « django-slack »^[12]. Cette librairie permet facilement de réaliser des commandes et d'envoyer des messages. Cependant, chaque message envoyé demande un fichier à lui seul, ce qui peut vite devenir encombrant dans l'architecture de l'application.

La seconde était d'utiliser l'API Slack^[13] directement et de faire des requêtes à celle-ci.

Il n'était pas voulu de pouvoir créer des commandes et vu qu'un système de requêtes était déjà mis en place en JavaScript, la version API a été choisie. Mon tuteur a donc créé l'application Slack avec tous les droits nécessaires et il ne restait qu'à utiliser le token* du bot pour réaliser les requêtes à l'API.

Voici (cf. [Figure 26](#)) la méthode implémentée pour les requêtes en Javascript :



```

// Method already implemented
function createRequest(url, method = "GET", body = {}, headers = {}) {
    headers["X-CSRFToken"] = csrftoken
    headers["Content-Type"] = "application/json"
    if (method === "POST") {
        return new Request(url,
            {method: method,
             body: JSON.stringify(body),
             headers: headers});
    }
    else {
        return new Request(url,
            {method: method,
             headers: headers});
    }
}

// API groups and methods can be found here : https://api.slack.com/methods
// For additional methods, APImethod should be "method1.method2.method3" ...
// Required body data can be found on the specific method page
function createBotRequest(APIgroup, APImethod, HTTPmethod, body = {}, headers = {}) {
    const url = `https://slack.com/api/${APIgroup}.${APImethod}`

    headers["Access-Control-Allow-Headers"] = ["X-CSRFToken", "Content-Type", "Authorization"]
    headers["Authorization"] = "Bearer xoxb-token"

    return createRequest(url, HTTPmethod, body, headers)
}

```

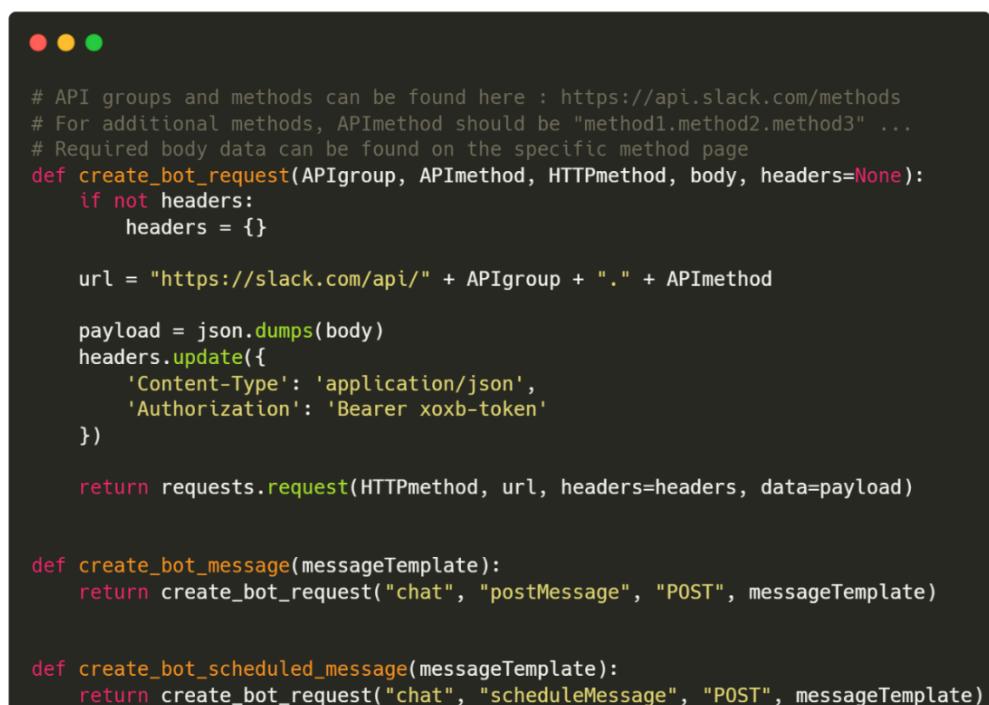
Figure 26 : extrait du code des requêtes à l'API Slack en JavaScript

Afin d'envoyer une requête à l'API, il faut renseigner la méthode voulue, son groupe et d'autres arguments obligatoires dans le body. Par exemple, pour envoyer un message dans un salon, le groupe est « chat », la méthode est « postMessage » et les arguments obligatoires sont « channel » (salon voulu), « text/attachments/blocks » et le token du bot. Ces éléments sont trouvables sur la documentation de l'API.

Lors de l'envoi de la requête vers l'API Slack, erreur CORS*. Cette erreur fait référence à la norme CORS, qui, lorsqu'une requête est effectuée depuis un navigateur client sur une origine distincte (un site avec un nom de domaine différent par exemple), différentes autorisations sont vérifiées par le serveur de cette même origine. Une première requête est alors envoyée au serveur distinct (« pre-flight request »), afin d'avoir plus d'informations concernant la requête et pour déterminer si l'envoi de celle-ci est sécurisé. Si c'est le cas, alors la vraie requête est envoyée au serveur, sinon, une erreur est levée.

Dans le cas de l'application, nous souhaitons faire une requête à l'API depuis un client, il faut alors fournir le token dans un en-tête « Autorisation ». L'ajout de cet en-tête oblige la requête à passer les vérifications CORS, or, l'API Slack ne permet pas de répondre correctement à la requête « pre-flight » donc la requête échoue.

Mon tuteur a proposé de passer sur Python pour faire les requêtes, de ce fait, les requêtes se feraient entre serveurs, seront plus sécurisées et n'auront pas à passer les vérifications CORS. Voici (cf. [Figure 27](#)) la nouvelle implémentation, réalisée en Python grâce à l'application Postman^[14], qui par ailleurs, aura permis de tester les requêtes API tout au long du développement :



```
# API groups and methods can be found here : https://api.slack.com/methods
# For additional methods, APImethod should be "method1.method2.method3" ...
# Required body data can be found on the specific method page
def create_bot_request(APIgroup, APImethod, HTTPmethod, body, headers=None):
    if not headers:
        headers = {}

    url = "https://slack.com/api/" + APIgroup + "." + APImethod

    payload = json.dumps(body)
    headers.update({
        'Content-Type': 'application/json',
        'Authorization': 'Bearer xoxb-token'
    })

    return requests.request(HTTPmethod, url, headers=headers, data=payload)

def create_bot_message(messageTemplate):
    return create_bot_request("chat", "postMessage", "POST", messageTemplate)

def create_bot_scheduled_message(messageTemplate):
    return create_bot_request("chat", "scheduleMessage", "POST", messageTemplate)
```

Figure 27 : extrait du code des requêtes à l'API Slack en Python

Cette requête fonctionne et ne provoque pas d'erreur CORS comme prévu.

Par ailleurs, d'autres fonctions utilisant la requête ont été créées, comme par exemple, la fonction « `create_bot_message` », qui envoie un message instantanément et la fonction « `create_bot_scheduled_message` », qui envoie un message en différé. Ces deux méthodes prennent un « `messageTemplate` » qui correspond au body voulu. Elles permettent aussi de ne pas avoir à répéter les paramètres nécessaires (« `chat` », « `postMessage` » ou « `scheduleMessage` », `POST`) lors de la création de requêtes.

Afin d'avoir un code plus clair, des méthodes auxiliaires ont été créées, contenant des body pré-faits (donc des contenus de messages à afficher par le bot) qui sont en réalité des dictionnaires. Ces dictionnaires sont facilement réalisables grâce à cet outil fourni par Slack : <https://app.slack.com/block-kit-builder/>.

Par la suite, un bouton a été ajouté lors de la création d'une revue de sprint, pour envoyer un message instantané indiquant que la revue est en ligne. Une autre implémentation a été réalisée, cette fois-ci, avec des messages différés indiquant de participer à la rétrospective. Deux messages différés sont créés automatiquement lors de l'activation d'un sprint et sont programmés pour être envoyés quatre jours et un jour avant la fin du sprint.

Voici le rendu final sur l'application Slack (cf. *Figure 28*) :



Figure 28 : exemple du message d'annonce de la revue

2.4 Manuel d'utilisation

Cette partie comporte des indications quant à l'utilisation des implémentations réalisées.

2.4.1 Application « Project Writing »

Tout d'abord, l'utilisateur peut accéder à la page en cliquant sur le module « Project Writing » de l'application « Scrum Companion ». Une fois sur la page, un tableau est généré, contenant les développeurs ainsi que les projets qui leur sont assignés actuellement.

L'utilisateur peut filtrer ce tableau en filtrant les développeurs par équipe ou les projets par état. Il peut également cliquer sur un projet pour être redirigé vers la page Jira de celui-ci ou bien, trier le tableau par nombre de projets croissant ou décroissant.

Lors du passage de la souris sur certains éléments, l'utilisateur peut obtenir des informations complémentaires, comme l'équipe du développeur ou le nom du projet.

Ensuite, en haut de la page se trouve un sélecteur contenant les différentes équipes de développement, l'utilisateur peut en choisir d'une à cinq et, en appuyant sur le bouton « Roll », lancer l'algorithme de recherche.

Les résultats de l'algorithme apparaîtront alors sur la droite des sélecteurs, si l'utilisateur n'est pas convaincu du résultat, il peut appuyer sur le bouton « Re-roll » qui relancera l'algorithme en excluant le résultat précédemment obtenu. Lorsqu'aucun résultat n'est disponible, un bouton « Reset request » apparaît, lors de l'activation de celui-ci, la liste des résultats exclus et les sélecteurs sont réinitialisés.

Ces résultats sont calculés de la sorte : les personnes sont des personnes appartenant aux équipes choisies dans les sélecteurs, par la suite, la personne ayant le moins de projets actuellement est choisie, en cas d'égalité, un facteur de criticité départage les personnes, celui-ci est calculé selon les états de leurs projets.

Enfin, en bas de la page se trouve un bouton permettant d'accéder à l'historique des projets réalisés sur les 365 derniers jours. L'affichage est identique à celui de l'autre tableau, cependant, l'utilisateur ne peut que filtrer les développeurs par équipe. En cliquant sur le bouton en bas de cette page, il peut retourner à la page principale.

Vous trouverez sur ce lien une vidéo de démonstration : [Démo-Project-Writing](#)

2.4.2 Preview de Review

À de l'approche d'une fin de sprint, les utilisateurs ayant les permissions d'administrateur peuvent créer une revue.

Cette revue peut, par la suite, être écrasée en cas de besoin de modification grâce à un bouton « Override Review ».

La création et l'écrasement d'une revue nécessitent d'avoir lié au moins un sprint Jira grâce aux sélecteurs présents au-dessus, auquel cas le bouton de création et d'écrasement n'apparaissent pas.

Vous trouverez sur ce lien une vidéo de démonstration : [Démo-Preview-Review](#)

2.4.3 Amélioration de la Review

Sur la page d'une revue de sprint, l'utilisateur peut voir une section « Projects ». Cette section est différente pour chacune des équipes et contient différents graphes, chacun relié à un projet travaillé durant le sprint par l'équipe en question. L'état du projet en général est aussi affiché pour permettre de voir l'avancée de celui-ci.

Le graphe est divisé en quatre

parties : les tâches terminées avant le sprint, durant le sprint, les tâches en cours et les tâches à faire. Au passage de la souris, l'utilisateur peut voir quel nombre de tâches chaque partie possède.

Vous trouverez sur ce lien une vidéo de démonstration : [Démo-Improved-Review](#)

2.4.4 Intégration Slack

Lors de l'activation d'un sprint, deux messages différés sont enregistrés automatiquement sur un bot Slack. Ces messages seront envoyés quatre jours et un jour avant la fin du sprint pour rappeler aux développeurs de participer à la rétrospective.

Par ailleurs, lorsqu'une revue est créée, un bouton « Notify Slack » est mis à disposition pour les utilisateurs bénéficiant des permissions d'administrateur. Ce bouton permet d'envoyer un message grâce au bot Slack pour notifier les développeurs que la revue est en ligne.

3 Projet « CampusPlaylist »

Dans cette partie, je vais présenter le second projet qui m'a été assigné : la création du jeu « CampusPlaylist ». Cette partie sera plus courte que la partie précédente, du fait qu'à l'heure actuelle où j'écris ce rapport, je travaille toujours sur le projet et seulement l'analyse a été réalisée.

Néanmoins, je présenterai tout de même les spécifications fonctionnelles et techniques qui m'ont été données ainsi que les analyses réalisées jusque-là, en montrant les architectures choisies pour le projet et la base de données.

3.1 Spécifications fonctionnelles et techniques

Vous retrouverez, sur [*Annexe 13 : Cahier des charges du projet CampusPlaylist*](#), le cahier des charges rédigé par mon tuteur concernant le projet du « CampusPlaylist ».

Pour ce projet, aucune application n'existe, je vais donc devoir partir de zéro pour réaliser le jeu. Cela implique que la conception, réalisation et validation de l'application sont entièrement de mon ressort. Le cahier des charges fourni donne des pistes sur certains choix de conception et sur les différentes fonctionnalités attendues.

Voici mon interprétation de celui-ci :

Pour ce qui est de l'architecture de l'application, le prototype de celle-ci fonctionnera également avec le Framework Django et une base de données PostgreSQL, comme le Scrum Companion.

Cependant, deux nouvelles choses sont demandées : la première est une API REST en Django, cette API permettra d'envoyer facilement des requêtes pour récupérer les données du jeu, la seconde est un Frontend en Angular, afin de remplacer le Frontend Django. De ce fait, l'application finale fonctionnera avec une API REST Django ainsi qu'un Frontend Angular, il faudra donc que j'apprenne ce nouveau Framework une fois le prototype fini.

Il est également précisé que l'utilisateur devra chercher ses musiques grâce à l'API Spotify, il faudra donc lier notre architecture à celle-ci.

Concernant les fonctionnalités demandées, le jeu possède plusieurs interfaces principales : La première interface est la page d'accueil, sur cette page aura lieu le jeu en lui-même, sous trois états. Chaque état représentera un tour de jeu :

Le tour zéro, représente le jeu inactif, aucune partie n'est en cours. Nous pouvons imaginer plusieurs boutons sur cette page, comme par exemple, un bouton pour lancer une partie ou pour accéder aux archives des dernières parties.

Le premier tour est le moment où les utilisateurs doivent proposer leur musique, cette musique doit être en raccord avec un thème précis, annoncé lors du début de la partie. Cette musique serait trouvable grâce à une barre de recherche, connectée à l'API Spotify.

Lors de l'affichage des résultats de la recherche, nous pourrions mettre une intégration de Spotify afin que l'utilisateur puisse écouter directement la musique depuis l'application, ou sinon, nous pourrions simplement réaliser une redirection vers l'application Spotify. L'utilisateur doit pouvoir sélectionner une musique parmi la liste, la modifier ou la supprimer.

À la fin du tour, une playlist serait créée grâce à l'API et toutes les musiques sélectionnées par les utilisateurs seraient ajoutées à celle-ci. Pour réaliser cela, il faudra trouver un moyen de connecter un compte à l'API afin de l'utiliser pour créer et stocker la playlist.

Le deuxième tour correspond au tour où les utilisateurs doivent parier sur les musiques. Un pari est une association entre un joueur et une musique de la playlist, le but étant de trouver le bon joueur qui a rajouté la dite musique.

Pour ce faire, on pourrait afficher la liste des musiques et, à côté de chaque musique, permettre à l'utilisateur de sélectionner, parmi la liste des joueurs, un joueur précis. Nous souhaitons éviter la possibilité de compléter une liste partiellement, de ce fait, un système de brouillon est demandé pour permettre au joueur de continuer plus tard, sans pour autant envoyer ses paris maintenant.

Un système de « j'aime » est aussi mis en place pour voter la musique favorite de la playlist. L'utilisateur ayant proposé cette musique aura une mention spéciale dans les résultats.

Lors de l'envoi des résultats finaux des paris, le joueur passe en mode spectateur. Ce mode spectateur ne permet de voir que la playlist et les joueurs. Un utilisateur n'ayant pas proposé

de musique au premier tour est automatiquement spectateur et ne peut donc pas réaliser de paris.

Nous pouvons potentiellement imaginer un dernier tour, comprenant les résultats des paris. Ces résultats seront des points incrémentés selon les paris gagnés ou perdus. Chaque joueur se verra alors attribuer un rôle selon ses points et ses exploits, voici la liste des rôles possibles :

- Le « finder » est la personne ayant eu le plus de points, le gagnant.
- Le « mentaliste » est la personne ayant eu le moins de points, le perdant.
- Les « autruches » sont les personnes qui ont été trouvées par une majorité de personnes.
- Les « caméléons », à l'inverse, sont les personnes qui n'ont été trouvées par quasiment personne.

Ces tours comprendront tous une date de début et une date de fin, de ce fait, une automatisation du jeu pourrait être réalisée pour passer automatiquement les tours, voire, créer automatiquement des parties.

La seconde interface est le profil de l'utilisateur, ce profil contiendrait les statistiques du joueur, comme par exemple, son nombre de parties jouées et son nombre d'acquisitions de chaque rôle (Finder, Mentaliste, Autruche, Caméléon). L'utilisateur pourra également être administrateur ou non.

Par ailleurs, j'ai pris l'initiative d'imaginer un système complémentaire qui serait un système de succès. Le joueur pourrait alors obtenir des médailles de différents niveaux en accomplissant des succès. Ces succès seraient affichés dans son profil, avec pour seule indication leur nom et le joueur devra alors découvrir les méthodes pour les obtenir. Par exemple, un succès « Finder » correspondrait au nombre de parties gagnées, avec pour paliers cinq parties (bronze), dix parties (argent), puis or et platine, etc.

Enfin, la troisième interface est la page des archives, cette page n'a pas réellement de descriptif sur le cahier des charges mais on peut imaginer un affichage par parties précédemment jouées. L'affichage comprendrait un podium avec les gagnants (les trois personnes avec le plus de points), des mentions spéciales par rôles (mentaliste, autruche et caméléon) ainsi que les joueurs et la playlist jouée durant la partie.

3.2 Rapport technique

Cette partie concerne la conception réalisée pour l'application, notamment la maquette, l'architecture et la base de données du projet.

3.2.1 Maquette du projet

Vous retrouverez, en [*Annexe 14 : Maquette du CampusPlaylist*](#), la maquette que j'ai réalisée pour l'application.

Tout à gauche se trouve la page d'authentification, l'utilisateur se connectera à Spotify pour accéder à l'application. Il faut encore que je réfléchisse concernant l'utilité d'une authentification par création de compte dédié à l'application.

Par la suite, en haut à gauche, se trouvent la page du profil et la page des succès. Le profil est accessible depuis une barre de navigation en haut de l'écran et contient : le nom et la photo de profil de l'utilisateur, les statistiques de celui-ci, les succès récents ainsi que l'historique des musiques qu'il a proposées.

Les succès seraient accessibles en cliquant sur la section « succès » de l'utilisateur, cette section mènerait à une page contenant tous les succès et les médailles associées. Aucune information complémentaire à part le nom du succès ne sera donnée, le joueur doit trouver lui-même comment obtenir un succès. L'obtention d'une médaille nécessite la médaille précédente et, en cliquant sur la médaille obtenue, cela afficherait le succès réalisé (ex : gagner dix parties).

À droite de la page des succès se trouve la page principale, avec le premier état, l'état inactif. Aucune partie est en cours, la date de la prochaine partie est affichée en haut et l'utilisateur peut soit voir les archives, soit créer une partie (seulement valable pour l'administrateur). La page « Main menu – Archive » apparaît lorsque nous voulons voir les archives, il est alors possible de parcourir par mois les différentes parties jouées, avec le podium, la musique favorite, les différentes mentions spéciales, la playlist entière et les joueurs.

La page de l'état de sélection des musiques se trouve sous la page d'état inactif. Celle-ci contient l'état précis du jeu (Song Selection), la date du prochain tour ainsi que le thème de la playlist, s'il existe.

Un bouton permet de sélectionner la musique à ajouter à la playlist. Ce bouton ouvre une page de recherche où l'utilisateur peut chercher une musique, l'écouter grâce à un embed* Spotify et la sélectionner. Par la suite, l'utilisateur peut soit, modifier sa musique en effectuant de nouveau une recherche, soit la supprimer s'il décide de ne plus jouer. De plus, un embed de la musique choisie est affiché afin de rappeler le choix du joueur.

L'état de la création des paris se trouve en bas à gauche de la maquette. Cet état contient encore une fois l'état précis du jeu, le thème et la date du prochain tour. Une liste de musique est affichée avec, à sa droite, deux icônes : la première permet l'ouverture d'une liste de joueurs, lorsque l'utilisateur clic sur un joueur, cela le sélectionne comme détenteur de la musique, la seconde permet de voter la musique en musique favorite de la playlist.

Une fois tous les choix complétés, l'utilisateur peut cliquer sur « Confirmer les choix » et les choix deviennent non-modifiables. Pour ce qui est du mode spectateur, pour les personnes ne pouvant pas jouer, nous affichons seulement la liste des joueurs et la playlist comme dit précédemment.

Enfin, la page finale avec les résultats de la partie sera similaire à une page des archives, avec les mêmes informations.

3.2.2 Architecture du projet

L'architecture du prototype demande l'utilisation du Framework Django, associé à une API REST Django. L'API REST sera donc implémentée en utilisant le Framework Django REST^[15] qui permet une mise en place simple de l'architecture.

De plus, pour la connexion à l'API Spotify, il a été proposé dans le cahier des charges, l'utilisation de la librairie Spotipy^[16], proposant des différentes fonctions pour accéder aux informations de l'API Spotify.

Le cœur de l'application résidera donc dans l'API REST Django, qui représentera notre backend. Ce backend sera connecté à un Frontend Djanjo, qui sera plus tard remplacé par un Frontend Angular, ainsi qu'à une base de données PostgreSQL.

Par la suite, nous aurons trois autres connexions :

La première, entre l'API REST et l'API Spotify, en passant par Spotipy, afin d'avoir des éléments relatifs au jeu, comme par exemple, la playlist sauvegardée.

La seconde, entre l'API REST et l'API Slack, cela n'est pas demandé de base mais étant donné que j'ai déjà travaillé avec l'API Slack lors du projet Scrum Companion, il serait intéressant de faire une intégration pour envoyer des messages relatifs aux états d'une partie par exemple.

La troisième, entre le Frontend Django et l'API Spotify, afin de récupérer des éléments relatifs à Spotify, comme la recherche de musiques en elle-même qui n'a pas de relation avec le jeu, en effet, seule la musique sélectionnée nous intéresse. Cette troisième connexion pourrait ne pas être possible en cas d'erreur CORS, comme vu dans le projet Scrum Companion.

Voici un schéma représentant cette architecture (cf. [Figure 29](#)) :

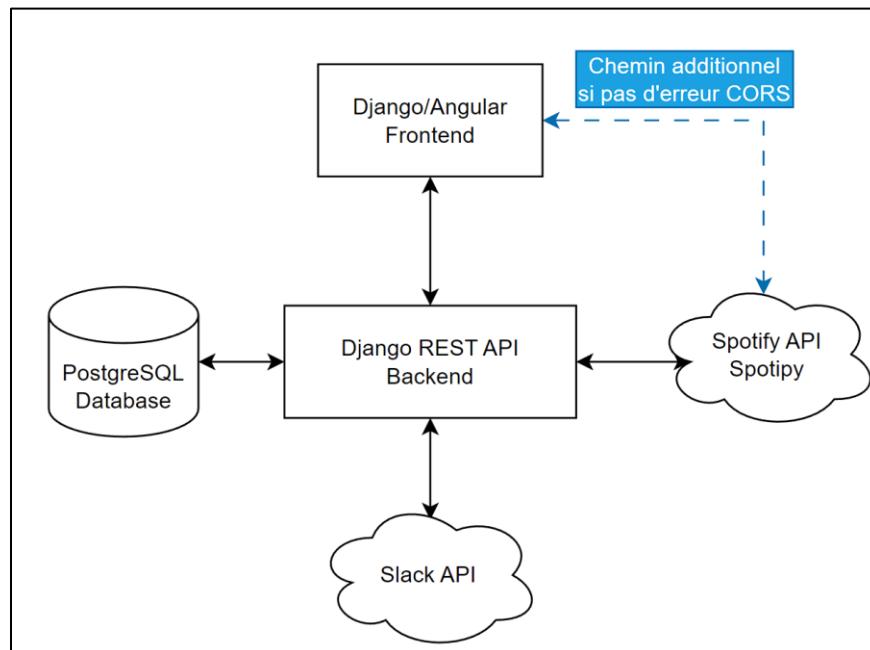


Figure 29 : architecture du projet

Cette création d'architecture me permet notamment de valider mon apprentissage en conception d'applications communicantes (apprentissage critique 3.2)

3.2.3 Architecture de la base de données

Un modèle entité-association est disponible sur [Annexe 15 : Modèle entité-association de la base de données du CampusPlaylist](#).

L'association « Game » est le centre de notre jeu. En effet, une partie commence à une date précise et nécessite une playlist, des utilisateurs, des musiques ainsi que des paris. Tous ayant un rapport entre eux, d'où l'association 4-aire.

Une musique est unique lors d'une partie, la contrainte CIF permet d'indiquer que, pour une playlist et un utilisateur donné, il ne peut y avoir qu'une musique, celle qu'il a choisie pour la playlist. Celle-ci a pour attribut un ID, un nom, un lien vers Spotify ainsi qu'une popularité, qui correspond au nombre de fois où cette musique a été favorite durant les parties.

Une musique est aussi contenue dans un pari. En effet, lors d'un pari, un utilisateur associe une musique à un autre utilisateur, de ce fait, la table « Guesses » a pour clés étrangères la musique et le joueur, et pour clé primaire, l'utilisateur associé à la musique.

L'utilisateur, quant à lui, possède un ID, un nom et un mot de passe. Étant donné qu'une connexion à Spotify est prévue pour accéder à l'application, le mot de passe sera potentiellement retiré au fil de la conception. L'utilisateur a la possibilité d'obtenir une médaille, à une certaine date. Cette médaille est associée à un succès, en effet, un utilisateur peut ne pas obtenir de médaille sur un succès et un succès n'être obtenu par aucun utilisateur.

Ces succès ont un ID, un nom, une description et un niveau, ce niveau correspond aux paliers possibles (bronze, argent, or, platine). Une autre conception avec un héritage était possible, dans le cas où un niveau de succès a une particularité. Par exemple, si la médaille d'or d'un succès offre un bonus sur l'application, cela aurait pu être renseigné dans la table « Gold » mais pas dans les trois autres tables. Pour l'instant, ce n'est pas souhaité.

Enfin, plusieurs connexions ont lieu entre une playlist et un utilisateur, celles-ci concernent les rôles. Une première conception était d'enregistrer, dans chaque partie (ou playlist) et dans chaque utilisateur, un dictionnaire contenant les statistiques, c'est-à-dire, combien de fois l'utilisateur a eu tel rôle ou qui étaient les rôles dans la partie (cf. [Figure 30](#)).

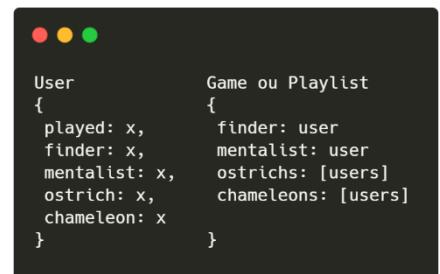


Figure 30 : exemple des dictionnaires de statistiques

Après discussion avec le client, celui-ci trouvait que la recherche d'informations, pour chaque utilisateur, serait potentiellement trop complexe, ou du moins, prendrait trop de ressources. En effet, pour récupérer les statistiques, il faut d'abord récupérer le dictionnaire, puis la clé précise, etc.

Pour régler ceci, une seconde conception a été réalisée, en créant des associations pour chaque rôle. De ce fait, on peut retrouver facilement tous les utilisateurs concernant un rôle dans une partie. Par exemple, pour avoir tous les utilisateurs « finders » de la partie concernant la playlist « A », il suffit de faire une requête et de récupérer tous les ID utilisateurs présents dans la table « Finder », qui ont pour ID playlist « A ».

Voici, par ailleurs, le schéma relationnel de cette base de données :

Playlists = (id_playlist, playlist_link);

Users = (id_user, name, password);

Songs = (id_song, name, song_link, popularity);

Achievements = (id_achievement, name, description, level);

Guesses = (id_user_guessed, #id_song, #id_user);

Game = (#id_playlist, #id_user, #id_user_guessed, game_date, #id_song);

Medal = (#id_user, #id_achievement, obtention_date);

Finders = (#id_playlist, #id_user);

Mentalists = (#id_playlist, #id_user);

Ostrichs = (#id_playlist, #id_user);

Chameleons = (#id_playlist, #id_user);

La conception d'une base de données relationnelle, à partir d'un cahier des charges, est l'un de mes apprentissages en BUT Informatique. Grâce à cette réalisation, je peux valider celui-ci (apprentissage 4.1).

4 Méthodologie et organisation

Dans cette partie, je vais expliquer mon organisation et mes méthodes de travail durant ce stage et je présenterai également quelques améliorations minimes réalisées hors projets.

4.1 Organisation du travail

Mon organisation personnelle était majoritairement concentrée sur un tableau Trello et un carnet. Dans le carnet, j'écrivais le brouillon de mes idées, que ce soient des interfaces, du code ou alors, des questions pour le client.

Par ailleurs, un daily meeting était organisé avec celui-ci pour voir l'avancée du projet, c'est majoritairement durant cette réunion journalière que je lui posais les questions écrites et que je lui faisais valider mes idées. Cela m'a permis, notamment, de mettre en œuvre mon apprentissage concernant l'instauration d'une démarche de suivi de projet (apprentissage critique 5.2) ainsi que ma capacité à rendre compte de mon activité professionnelle et de mes réalisations (apprentissage critique 6.2).

Concernant l'organisation du code, j'ai accès à un dépôt Bitbucket*, où le projet est hébergé. Cette version du projet est la version de développement, différente de celle de production, et se trouve sur une branche « master ».

Lorsque je travaillais sur le projet « Scrum Companion », je devais réaliser une branche par projet modifié, nommée avec le nom de celui-ci. De ce fait, j'aurais dû avoir cinq branches (quatre pour les mini-projets du rapport technique et une pour les modifications hors-projet) mais n'ayant jamais utilisé les branches auparavant, j'ai réalisé trois projets sur une même branche et seulement trois branches ont été créées.

Par la suite, une « pull request » était demandée pour chacune de ces branches, afin que mon tuteur puisse observer le code réalisé et me dire les problèmes et modifications à faire, avant d'appliquer les changements sur la branche « master ».

Enfin, j'ai veillé à ce que mon code soit compréhensible et documenté, c'est pourquoi j'ai réalisé des commentaires sur l'entièreté de mes implémentations, afin d'expliquer mes choix et mon code aux futures personnes travaillant sur l'application.

4.2 Méthodes de travail

Durant mon stage, je ne possédais pas d'équipe, de ce fait, j'étais entièrement autonome sur ce que je produisais. Cette autonomie s'est surtout ressentie lors des trois premières semaines de stage où mon tuteur n'était pas présent pour cause de congés.

Lors de cette période, ma démarche principale était d'avancer sur le projet, même sans retour client, afin d'avoir une architecture déjà bien avancée et pour, par la suite, modifier celle-ci selon ses besoins et selon les erreurs réalisées. Pour trouver mes informations, j'ai majoritairement utilisé la documentation Django^[8] ainsi que le forum StackOverflow^[17] en cas de blocages et de bugs.

J'ai également pris l'initiative de réaliser des améliorations non demandées, de proposer des solutions alternatives et de ne pas suivre complètement le cahier des charges afin de me donner une certaine liberté. Par exemple, lors de l'application du DRY sur le code de la revue ou encore, lors de la réalisation de la [Preview de Review](#), où j'ai décidé d'écraser la revue elle-même, plutôt que de suivre le cahier des charges et de créer une revue sans enregistrer ses données dans la base de données. Le client a, par ailleurs, beaucoup aimé cette idée.

4.3 Réalisations minimes

Plusieurs problèmes ont été repérés sur l'application Scrum Companion lors de mon analyse de celle-ci, j'ai donc pris l'initiative de résoudre ces problèmes.

Tout d'abord, sur le module « Support Issues », une balise « div » était manquante dans l'architecture HTML, de ce fait, l'affichage n'était pas correct et cassait la page.

Ensuite, sur le module « Gantt », il y avait un tableau à trois colonnes, la première et deuxième possédaient des flèches pour être triées. Or, ces flèches étaient mal positionnées et se trouvaient à l'opposé du tableau. Cela était majoritairement dû au CSS des intitulés de colonnes qui n'étaient pas en position « relative ».

Enfin, une division par zéro pouvait avoir lieu lors de la sauvegarde d'un sprint où certaines story n'avaient pas de points. Pour régler ceci, j'ai simplement rajouté des vérifications dans le cas où il n'y avait pas de points, si c'est le cas, nous renvoyons zéro.

Conclusion

Pour conclure, ce stage a grandement été bénéfique pour moi.

Mes six compétences de BUT Informatique ont été mises à l'épreuve durant celui-ci, en réalisant du développement, de l'optimisation, de l'infrastructure et de la communication entre serveurs, de la base de données, de la gestion de projet ainsi que de la communication professionnelle.

Mon apprentissage principal durant ce stage était la gestion des différentes API. Lors de mes cours, j'avais énormément de lacunes sur ce sujet. Utiliser ces APIs dans un contexte réel était pour moi un moyen de comprendre leur fonctionnement et de rattraper, voire supprimer, ces lacunes.

Par la suite, j'ai pu également reprendre ce que j'ai appris en cours, comme par exemple, la création de base de données, pour le projet CampusPlaylist (apprentissage critique 4.1) ; la mise en place d'un code clair, respectant les normes de programmation, lors du refactor de la page de la revue pour que celle-ci respecte la norme DRY (apprentissage critique 1.2) ou encore, la gestion d'un projet avec un véritable client, j'ai su mettre en place une démarche de suivi de projet avec celui-ci, pour obtenir ses besoins, ses avis sur les prototypes ou encore, sa validation des différentes réalisations (apprentissage critique 5.2).

Je suis très fière de ce que j'ai pu produire durant ces neuf premières semaines de stage.

Par ailleurs, les améliorations réalisées pour le projet Scrum Companion sont déjà mises en production. Ces implémentations seront à présent utilisées par toutes les équipes techniques de GoodBarber, un premier retour très positif des utilisateurs a déjà eu lieu, surtout concernant la partie [Amélioration de la Review](#). Cette mise en production me permet notamment de valider un apprentissage critique de la troisième année de BUT Informatique, l'apprentissage « Intégrer des solutions dans un environnement de production ».

À présent, mon objectif pour les trois dernières semaines de ce stage est de continuer l'application CampusPlaylist, afin de potentiellement réaliser le premier prototype.

Vu le 14/06/2023,
Sergio MIRANDA CARVALHO

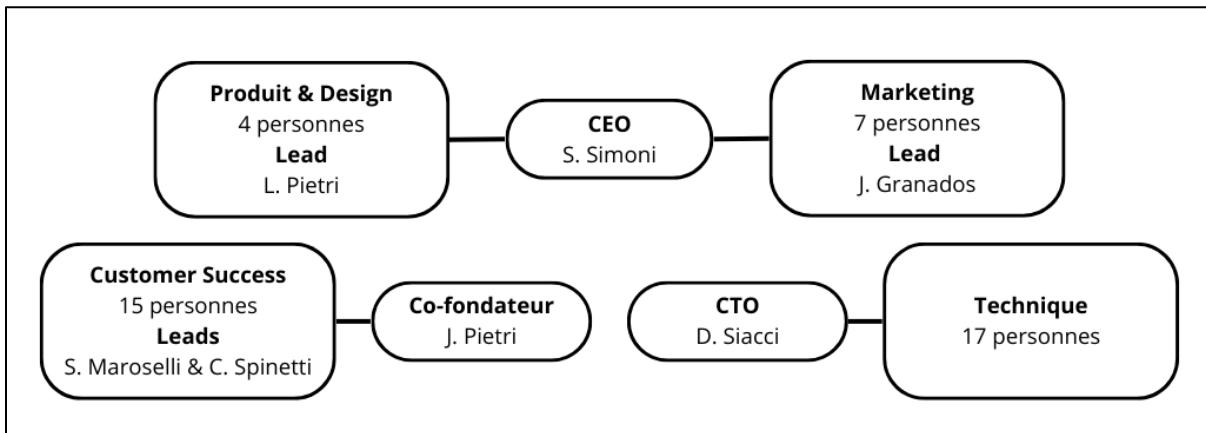


Bibliographie

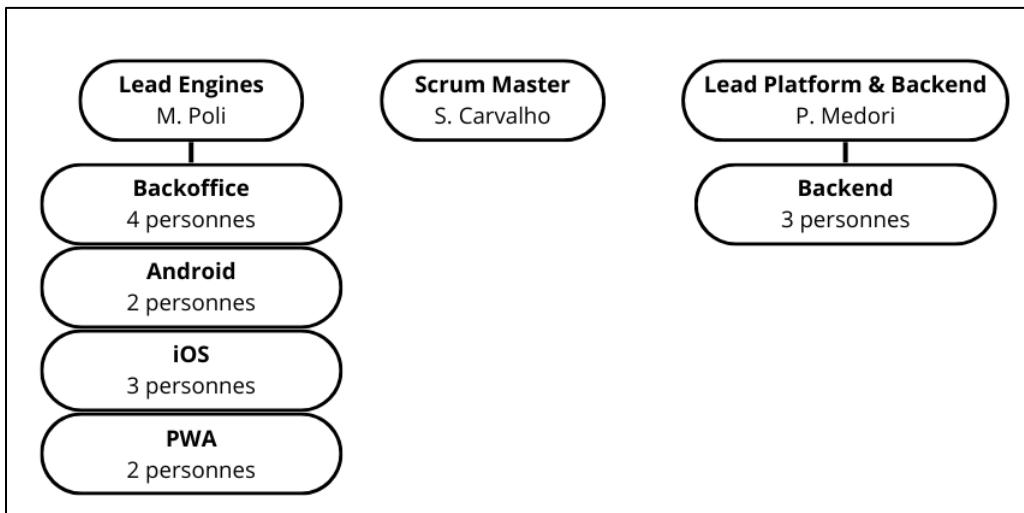
- [1] « Créateur d'applications mobiles Android, iOS sans coder - App Builder français ». <https://fr.goodbarber.com> (consulté le 14 mai 2023).
 - [2] « GoodBarber - Combien coûte la création d'une application ? » <https://fr.goodbarber.com/pricing/> (consulté le 7 mai 2023).
 - [3] Atlassian, « Logiciels de collaboration pour les équipes de développement, informatiques et métier », *Atlassian*. <https://www.atlassian.com/fr> (consulté le 14 mai 2023).
 - [4] Atlassian, « OKR : le guide ultime », *Atlassian*. <https://www.atlassian.com/fr/agile/agile-at-scale/okr> (consulté le 12 mai 2023).
 - [5] « Django tutorial », *Django Project*. <https://docs.djangoproject.com/en/4.2/intro/tutorial01/> (consulté le 17 avril 2023).
 - [6] « Débutez avec le framework Django », *OpenClassrooms*. <https://openclassrooms.com/fr/courses/7172076-debutez-avec-le-framework-django> (consulté le 15 avril 2023).
 - [7] « CodingEntrepreneurs - YouTube ». https://www.youtube.com/watch?v=SIHBNXW1rTk&list=PLEsfXFp6DpzRMby_cSoWTFw8zaMdTEXgL (consulté le 20 avril 2023).
 - [8] « Django documentation », *Django Project*. <https://www.djangoproject.com/> (consulté le 20 avril 2023).
 - [9] « Elite Admin Template - Landing Page ». <http://eliteadmin.themedesigner.in/demos/bt4/landingpage/index.html> (consulté le 14 mai 2023).
 - [10] « How To Sort a Table ». https://www.w3schools.com/howto/howto_js_sort_table.asp (consulté le 28 avril 2023).
 - [11] « Select2 - The jQuery replacement for select boxes ». <https://select2.org/> (consulté le 9 juin 2023).
 - [12] « django-slack ». <https://chris-lamb.co.uk/projects/django-slack> (consulté le 20 avril 2023).
 - [13] Slack, « Unlock your productivity potential with Slack Platform », *Slack API*. <https://slack.com> (consulté le 6 juin 2023).
 - [14] « Postman API Platform », *Postman*. <https://www.postman.com> (consulté le 6 juin 2023).
 - [15] « Django REST framework ». <https://www.django-rest-framework.org/> (consulté le 12 juin 2023).
 - [16] « Spotify - Documentation ». <https://spotipy.readthedocs.io/en/2.22.1/> (consulté le 12 juin 2023).
 - [17] « Stack Overflow - Where Developers Learn, Share, & Build Careers », *Stack Overflow*. <https://stackoverflow.com/> (consulté le 11 juin 2023).
-

Annexes

Annexe 1 : Organigrammes de l'entreprise

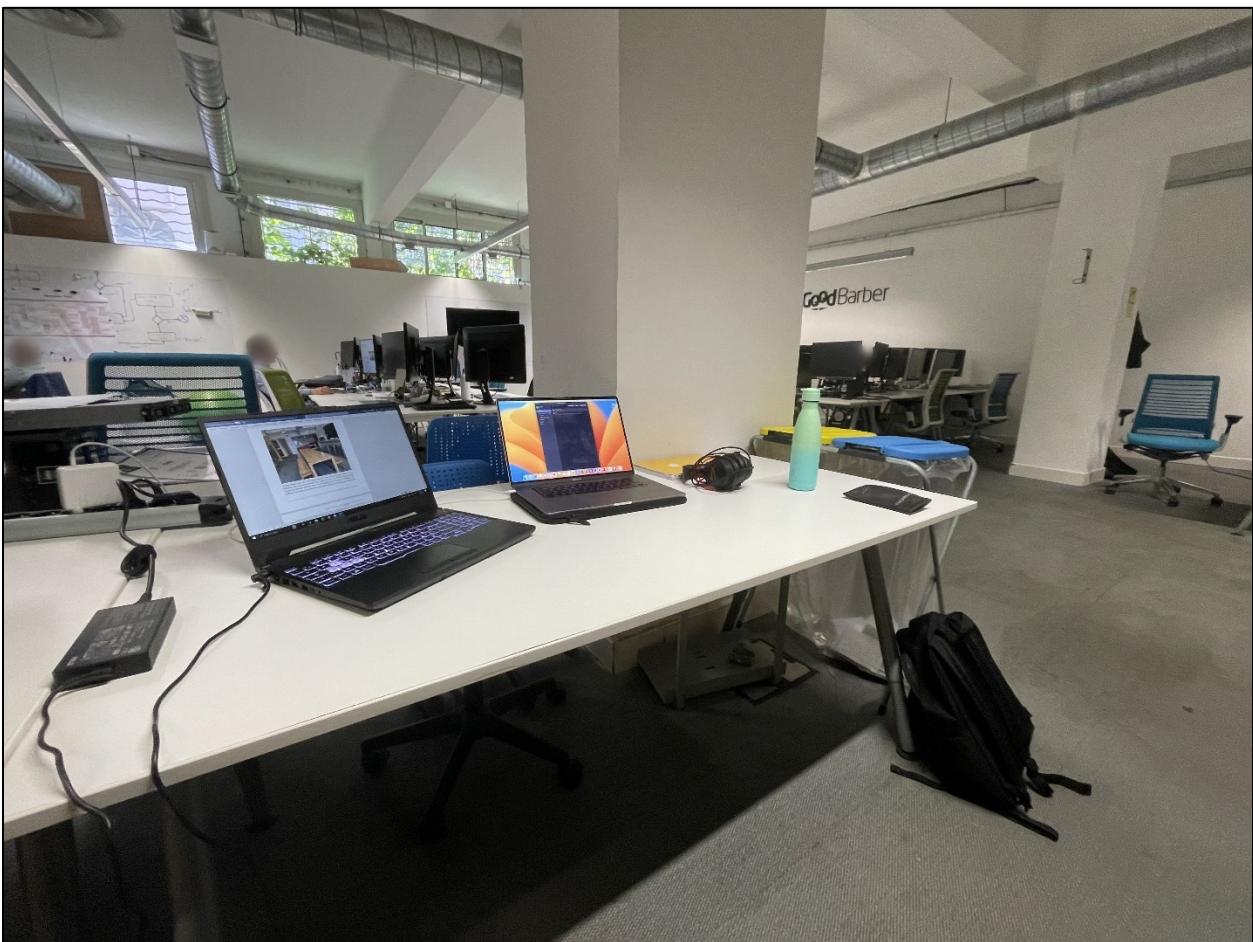


Organigramme de l'entreprise avec les différentes équipes ainsi que leurs chefs et patrons



Organigramme de l'équipe Technique avec les différents chefs et sous-équipes

Annexe 2 : Open-space et bureau personnel



Annexe 3 : Aperçu du module de Sprint

Sprint

stella Logout

Active Sprint

Test-sprint
June 15, 2023, 10 a.m. – June 26, 2023, 2:30 p.m.

Goals
Yeah, well, we have to begin somewhere

TO PUBLISH
todo Magic palette

COMPLETE DEV
todo Custom fonts in GB App

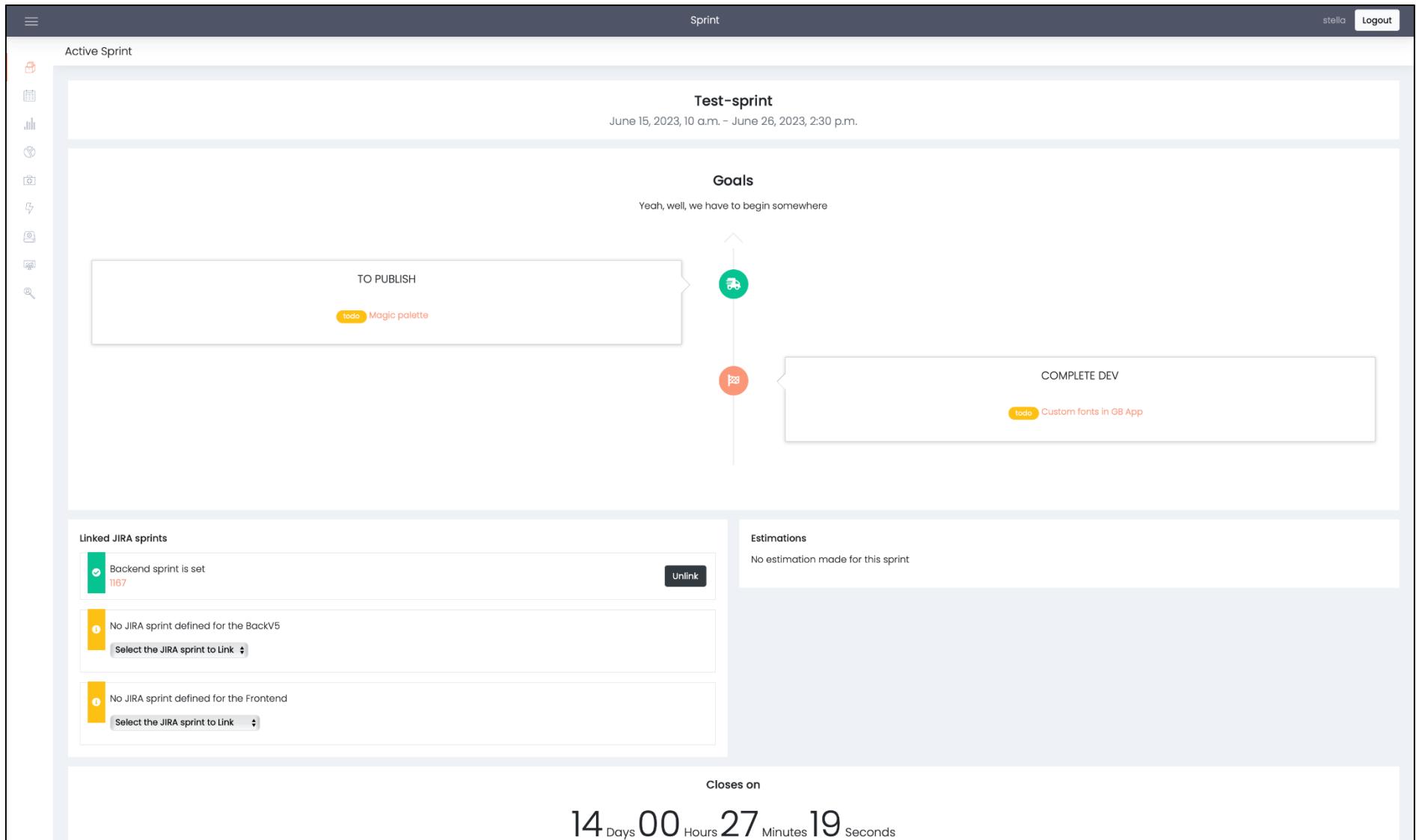
Linked JIRA sprints

- Backend sprint is set 1167 [Unlink](#)
- No JIRA sprint defined for the BackV5 [Select the JIRA sprint to Link](#)
- No JIRA sprint defined for the Frontend [Select the JIRA sprint to Link](#)

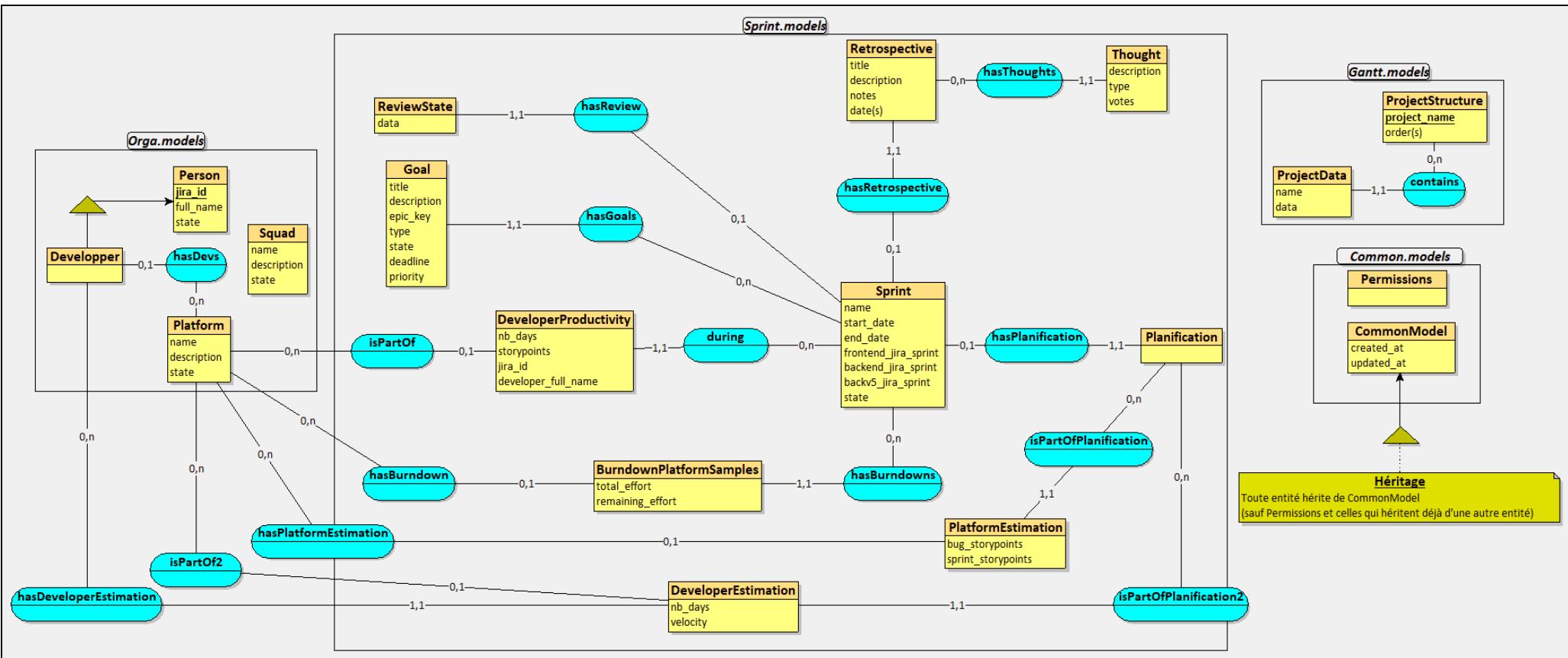
Estimations
No estimation made for this sprint

Closes on

14 Days 00 Hours 27 Minutes 19 Seconds



Annexe 4 : Modèle Entité-Association



Annexe 5 : Cahier des charges du projet Scrum Companion

Début de stage Stella-Maria

1/ L'écriture des projets

Dans le cadre de travail de GoodBarber, il est demandé aux développeurs de participer à l'élaboration des spécifications techniques. Le développeur en charge d'écrire la spécification technique est aussi responsable des tests techniques.

L'équipe gère en simultané une trentaine de projets et il est parfois difficile de se rappeler qui à quel projet en charge, en cours d'écriture, ou en cours de test. Il serait très utile d'avoir un outil résumant cette information afin de voir quel développeur serait le plus disponible pour prendre en charge un nouveau projet.

L'objectif du stage sera de répondre à cette problématique en ajoutant à l'outil « Scrum Companion » les indicateurs de prise en charge des projets par les développeurs. Il sera demandé au stagiaire d'être force de proposition pour le design de la fonctionnalité.

Description du besoin

Nous allons travailler sur 3 aspects de la charge en "projet" par les développeurs :

- La charge courante

Il s'agit de dresser la liste des développeurs et d'y associer la liste des projets en cours (phase d'écriture, en attente, phase de développement). Cela permettrait de voir qui a la charge de projet la moins élevée et qui pourrait, à priori, prendre en charge au mieux un nouveau projet.

Chaque ligne de la liste devra afficher le nombre de projets par phase, ainsi que le nom du projet. Il faudra également afficher, pour chacun des projets, le nombre de stories "à valider" associées.

La liste de développeurs devra être ordonnée par le nombre de projets décroissant.

➤ Historique sur l'année

Le but est de lister les développeurs et d'y associer les projets publiés sur l'année. Cela peut permettre de voir la typologie des projets et les équipes les plus impactées par la prise en charge des projets. La liste sera similaire à la liste précédente.

➤ Choix aléatoire d'un chef de projet

Il n'est pas rare pour le Scrum Master de poser la question suivante : "Qui veut prendre en charge ce projet ?". Il s'en suit généralement un silence ou seules les mouches sont audibles. Dans ces cas-là, il serait intéressant d'avoir un script capable de déterminer une personne pour l'écriture d'un projet.

Ce script devra prendre en entrée certains critères et en sortira la meilleure personne pour prendre en charge un projet.

Les critères d'entrée sont :

- La charge des développeurs
- Les plateformes susceptibles d'écrire ce projet
- Criticité du projet (Bonus)

L'algorithme devrait alors désigner un des développeurs d'une plateforme en particulier qui est le moins chargé actuellement en termes de projets.

- Bonus
- Afficher, dans la liste de charge courante et historique, des personnes qui ne sont pas forcément des développeurs.
- Gérer une criticité de projet, celui-ci ne serait accordé qu'à une personne avec une certaine seniorité de gestion de projet.

Note : afin d'assurer l'essentiel de la fonctionnalité fonctionnant à l'issue du stage, il sera demandé de faire un "plan de bataille" avec le tuteur de stage.

2/ Améliorations de la review

Actuellement, nous sommes capables d'avoir une vue du travail accompli par les différentes équipes dans un sprint.

Nous aimerais ajouter dans cette review, la liste des projets sur lesquels chaque équipe est intervenue.

Il faudra être force de proposition pour l'UX et également pour trouver les infos pertinentes à afficher (nom du projet, état du projet pour l'équipe ?).

3/ Intégration avec Slack

Il serait intéressant que lorsqu'une review de sprint est disponible, d'envoyer un message dans le channel #tech-general indiquant que le sprint est terminé et que la review est disponible (un lien vers la review serait également intéressant).

Par la suite, on peut imaginer d'autres fonctionnalités pour animer le sprint à divers moments : des rappels pour participer à la rétrospective, des points sur l'état des sprint et/ou des projets, des phrases culte sorties de films français chaque matin...

4/ Preview de review

Il arrive souvent de devoir préparer le sprint et comprendre les grandes tendances, avant d'avoir une review finale.

Il faudrait être capable de faire une preview sans avoir à enregistrer les données.

Annexe 6 : Croquis de conception de l'application « Project Writing »

à mettre avant tout

filter commun ?

APERÇU PAGE

CHARGE COURANTE

Équipes :	IOS	ANDROID	BACKEND	...
EtatProjet :	En Attente	En Attente	En Attente	...
Developpeurs :	Nombre projets	Projets assignés		
Nom Dev 1	2	proj (5)	proj (6)	
Nom Dev 2	1	proj (1)		No Stories
Nom Dev 3	0			

voulez la mise en page des projets

(buttons, selectionne...)

pas une de la colonne, à voir comment la placer

HISTORIQUE

voir quelles filtres à mettre

filtrer par années ?

Bande des filtres

Developpeurs	No Proj	Projets terminés
Nom Dev 1	5	proj 1, proj 2, proj 3, proj 4, proj 5
Nom Dev 2	3	proj 1, proj 2, proj 3
Nom Dev 3	0	

Limites les listes ? (sinon +50 items pour 1 tableau)

↳ soit on limite à 10, ou ex 10 qui m'reste de charge

↳ soit on fait des paginations pour les tableaux

1 12 13 ...

informations sur le projet

affichage d'une carte du projet

Carte de projet (encliquant sur un projet)
(au juste & redimensionnée la page?) nb stories ou nb effaçais

RECHERCHE CHÉF DE PROJET

Type :

-
-
-
-
-
- Voir

Équipe :

-
-
-
-
-
- IOS
- ANDROID
- BACKEND
- BACKOFFICE
- PWA

lien vers le projet

Historique après recherche ?

↳ filtres communs tel que nom du projet

↳ duplique les filtres dans les 2 sections

(historique et charge)

Si type = liste (choix possible)

Si type = unique (choix impossible)

Nom Dev Charge

Nom Dev Charge

Nom Dev Charge

Annexe 7 : Maquette Figma de l'application

The wireframe illustrates a user interface for a 'Project Leader' application. The top navigation bar includes a menu icon, the title 'Project Leader', a user name 'stella', and a 'Logout' button. On the left, there is a vertical sidebar with icons for file management, calendar, charts, developer tools, and project status.

The main content area is titled 'Current load'. It features a 'Filters' section with two rows of buttons:

- Platforms:** iOS (dark grey), Backend (yellow, selected), Backoffice (pink), PWA (light pink), Android (green), None (grey).
- States:** Ecriture (blue), En attente (purple), Développement (dark purple).

Below the filters is a table titled 'Developer' showing project statistics:

Developer	Nb projects	Current projects
Developer's name	5	Project's name 9, Project's name 7, Small 5 Project's name 4, Very long project with a big name 2
Developer's name	2	Project's name 5, Project's name 1
Developer's name	0	

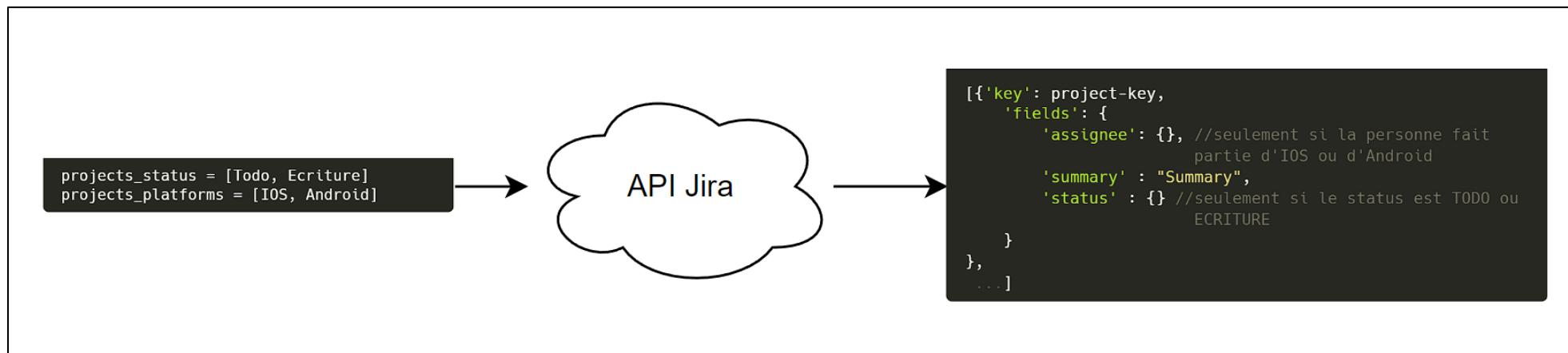
Navigation arrows (left, right) are located at the bottom of this section.

At the bottom of the page is a 'Project Leader finder' section:

Platform(s): Roll

Developer's name: 0 Contact Re-Roll

Annexe 8 : Entrées et sorties de l'API Jira



Annexe 9 : Cycle de vie d'un projet Jira

Un projet Jira peut se trouver dans trois tableaux principaux différents, le tableau IB, EB ou GTA.

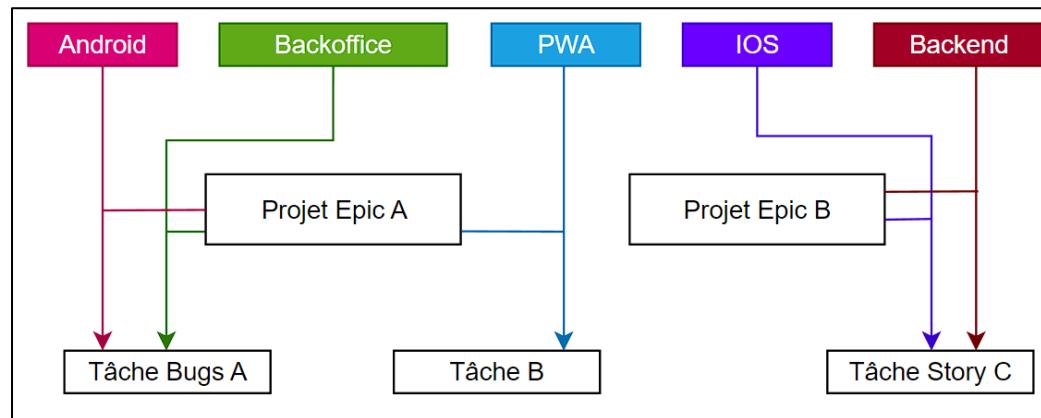


Le tableau IB contient les projets en cours de spécifications, que ce soit de la spécification produit ou technique, c'est ici que les projets « en cours d'écriture » sont. Ces projets doivent par la suite être validés, auquel cas ils ne pourront pas passer en EB.

Le tableau EB contient tous les projets en cours de développement. Ces projets suivent les spécifications rédigées auparavant, de ce fait, pour revenir sur le facteur de projet de l'algorithme de chef de projet, un projet dans EB est aura une charge plus faible qu'un projet dans IB car il ne fait que suivre ce qui a déjà été rédigé et préparé.

Le tableau GTA contient les projets archivés qui sont déjà en production.

Par ailleurs, un projet (Epic) est un regroupement de tâches. Celui-ci peut être travaillé par une ou plusieurs équipes. Une tâche d'un projet est liée à une équipe, par exemple, l'équipe Android et l'équipe Backoffice ont la tâche « Bugs A », néanmoins, ils n'auront pas forcément les mêmes choses à faire sur celle-ci.



Annexe 10 : Résultats de l'algorithme de recherche

The screenshot shows a user interface titled "Project Leader Finder". At the top, there is a search bar with the text "Platforms: iOS x PWA x". To the right of the search bar is a large orange button labeled "Roll". Below the search bar, the name "Bizzari Carla" is displayed next to a profile picture, with the number "1" indicating the count of results. There are two buttons on the right: "Contact" and "Re-Roll".

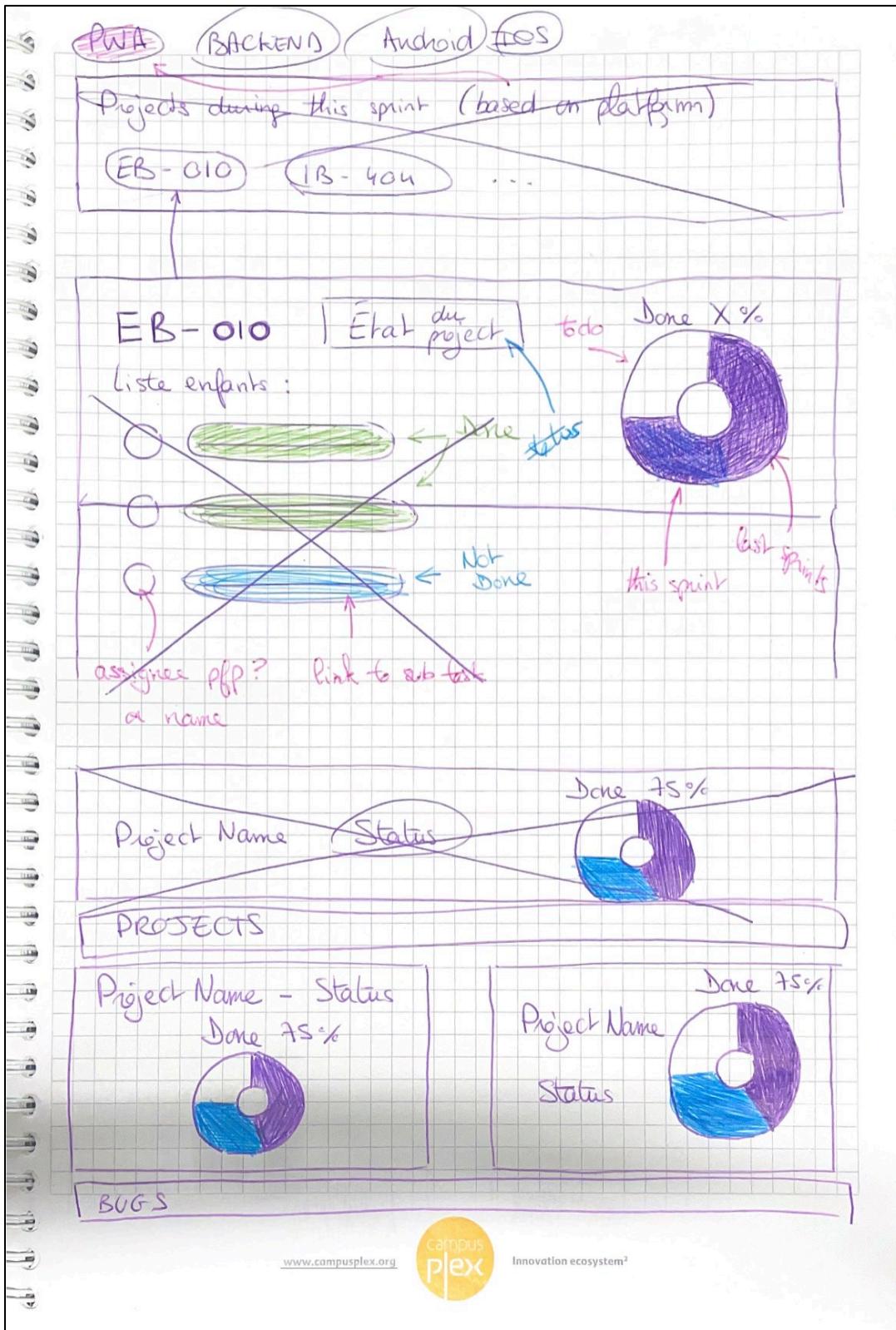
Annexe 11 : Page de l'application Project Writing

The screenshot shows a dashboard titled "SCRUM Companion". On the left, there is a sidebar with various icons. The main area is titled "Project Leader Finder" and displays a "Current Load" section. It includes a "Filters" section with dropdowns for "Platforms" (selected: iOS) and "States" (selected: To do). Below this is a table titled "Persons" showing current projects assigned to team members:

Persons	Nb projects	Current projects
Lisa Colombani	5	EB-660 (25), EB-659 (25), EB-12075 (0), EB-768 (0), EB-563 (0)
Jim Marchetti-Ettori	2	EB-760 (18.5), EB-764 (18)
Flo Lucioni	2	EB-763 (21), EB-747 (13)
Mathieu Fancello	0	
Sergio Carvalho	0	
Alexandre Carvalho	0	

A green notification at the bottom right says "Someone has been found !".

Annexe 12 : Croquis de conception de l'affichage des projets



Annexe 13 : Cahier des charges du projet CampusPlaylist

The Playlist Game

1/ Architecture

L'idée est de baser le système sur le Framework Django. Le prototype complet sera réalisé avec Django + Rest API. Nous le considérerons comme API + Frontend app (Angular) dans le futur.

Nous utiliserons PostgreSQL pour la base de données.

2/ Caractéristiques

États du jeu

Le jeu aura 3 états :

- Pas de partie en cours :

C'est à ce moment que les utilisateurs pourront consulter les résultats de la dernière partie, des archives ou encore, de leurs profils.

- Choix des musiques :

C'est à ce moment que les utilisateurs devront choisir une musique. Ils auront trois actions possibles : ajouter, modifier ou supprimer la musique.

La musique pourra être cherchée grâce à l'API Spotify, en renseignant le nom ou l'URL de celle-ci. Les utilisateurs pourront avoir une prévisualisation de la musique.

➤ Création des paris :

Les utilisateurs devront remplir un formulaire où ils associeront un utilisateur à une musique. Différents états sont possibles selon le statut du joueur :

- Le joueur n'a pas choisi de musique lors de la dernière phase : celui-ci n'a accès qu'à la playlist et aux joueurs.
- Le joueur a choisi une musique mais n'a pas rempli le formulaire : celui-ci a accès à la playlist, aux joueurs et peut remplir le formulaire, en créant un brouillon ou en envoyant une réponse définitive.
- Le joueur a envoyé une réponse définitive : celui-ci n'a accès qu'à la playlist et aux joueurs, un message indique qu'il a déjà joué et qu'il doit attendre la date des résultats.

Les dates des prochaines parties ou des prochains rounds doivent être présentes lors de tous les états.

Accueil

En arrivant sur le site, l'état actuel du jeu sera affiché, avec les caractéristiques associées.

Compte du joueur

Pour jouer, l'utilisateur se doit d'avoir un compte. Celui-ci sera créé par un administrateur.

L'utilisateur aura accès à son profil, où il pourra voir ses statistiques.

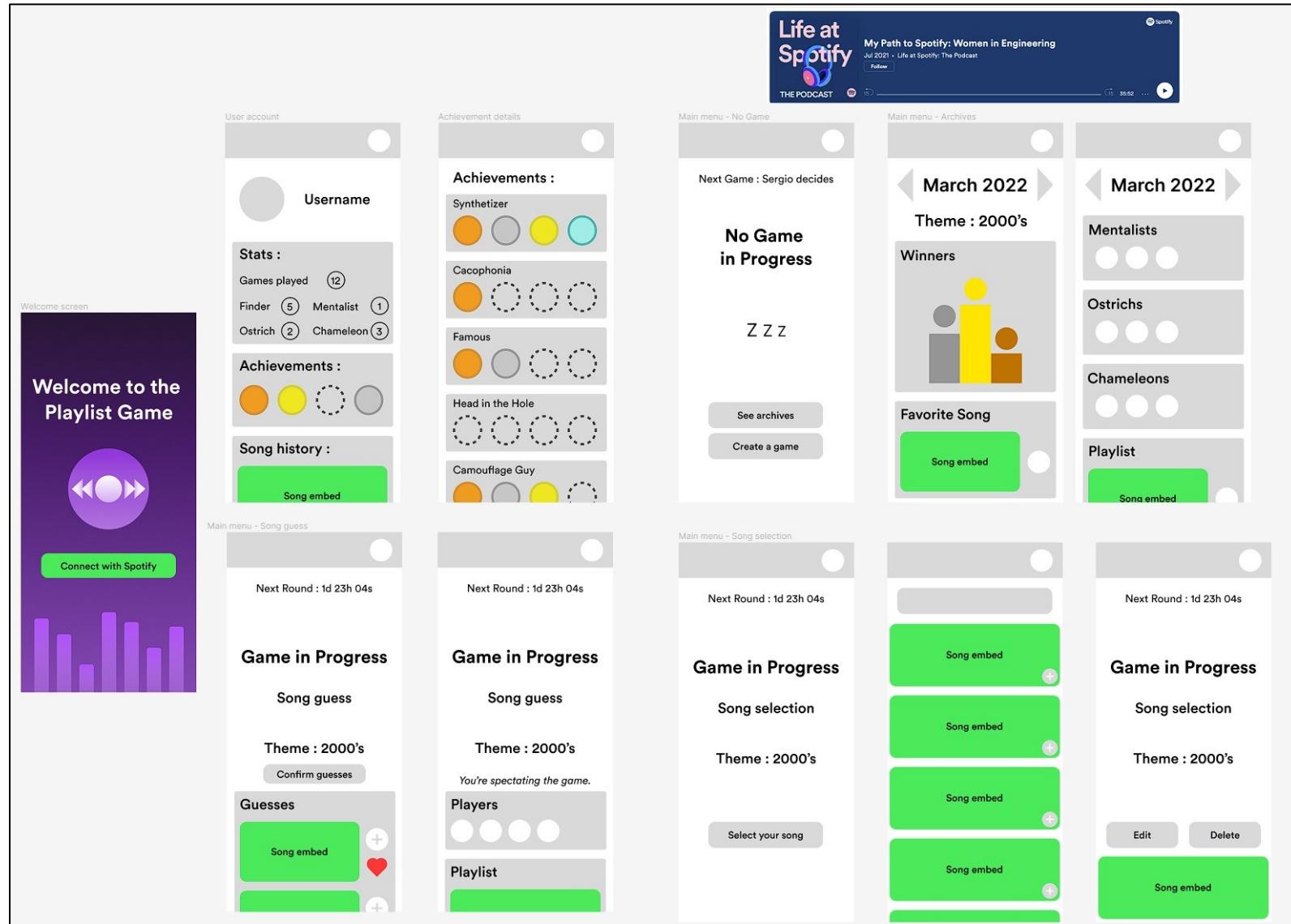
Archives

3/ Ancien prototype Figma

4/ Méthodes de Python API pour aider

5/ Ancien modèle entité-association

Annexe 14 : Maquette du CampusPlaylist



Annexe 15 : Modèle entité-association de la base de données du CampusPlaylist

