Compte Rendu d’activité intermédiaire

– Premier Semestre (S3)–



**Lucas LEVEQUE, Berachem MARKRIA & Joshua LEMOINE**

17/12/2022

BUT2 Informatique

# INTRODUCTION

Ce premier compte rendu d’activité se concentre sur la mise en place du moteur de jeu en C et de la création des divers dictionnaires demandés en JAVA. La base de données et le développement web n’ont pas été oubliés mais leurs sujets n’étant pas encore disponibles ou très récemment disponibles, il est compliqué d’avancer sur ces derniers. Enfin, nous avons mis en place un espace de travail partagé afin de faciliter le transfert d’informations dans l’équipe.

Nous avons donc réussi à totalement terminer la partie JAVA. Le travail a été considérable et nous sommes très satisfaits du résultat. Concernant la partie C à réaliser : nous avons presque terminé. Il reste néanmoins quelques parties que nous devons corriger afin que celles-ci fonctionnent.

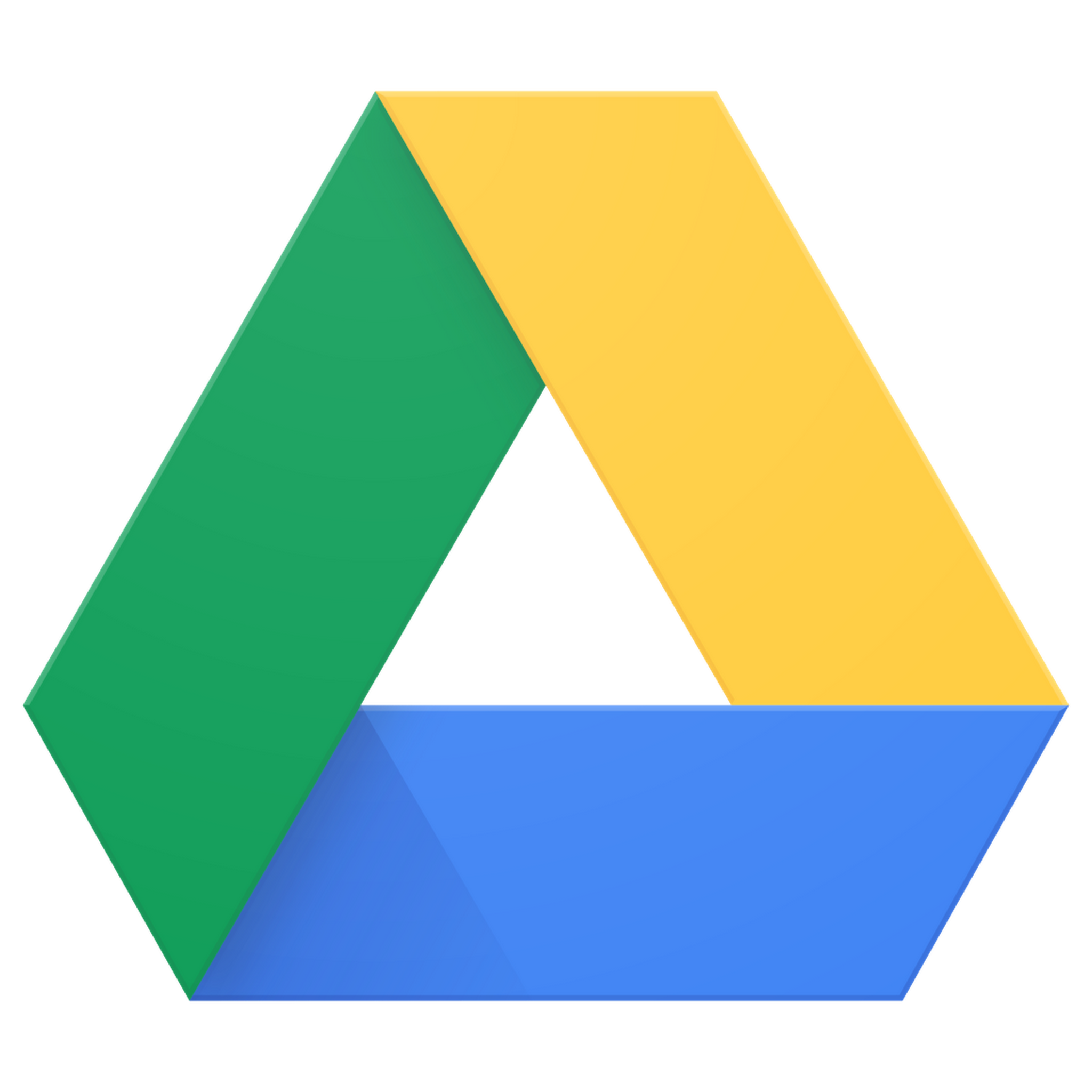
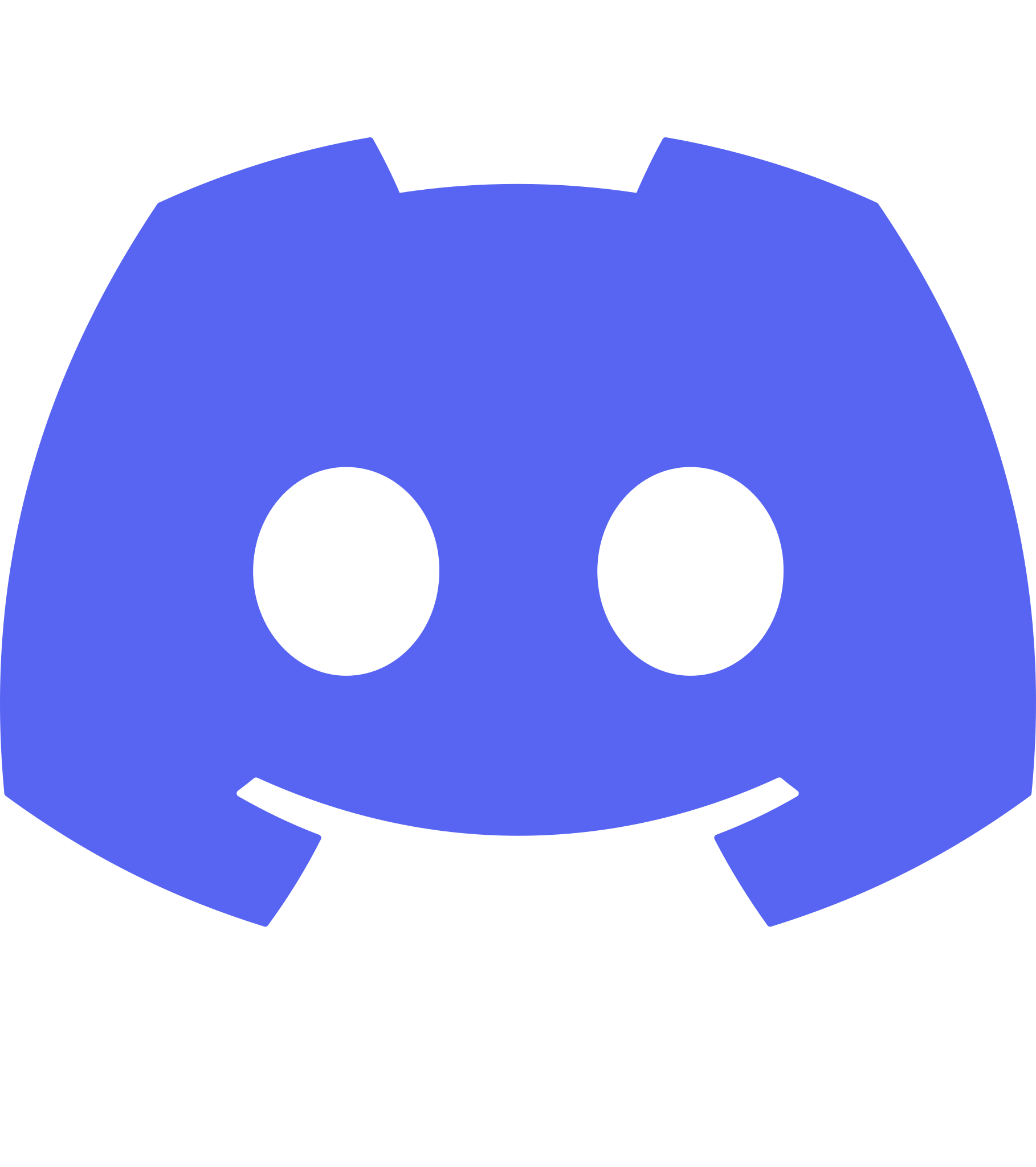
Au sujet de la base de données, nous avons pour le moment réalisé un très rapide schéma. Toutefois, il est encore nécessaire de procéder à de nombreuses modifications pour lui donner sa forme finale. Le sujet étant arrivé le 16/12/2022 nous n’avons que légèrement commencé ces modifications mais avons sérieusement avancé sur la réflexion.

A ce jour, n’ayant aucun sujet pour le développement web aucune avancée n’a été effectuée sur ces points.

Ainsi, nous sommes en bonne voie et pensons être en mesure de terminer le travail demandé.

# OUTILS DE COLLABORATION

Nous avons créé un [dépôt Github](https://github.com/Berachem/Zoo-ggle) privé pour héberger le code source et les documents de projet. Nous utilisons également [Drive](https://drive.google.com/drive/folders/1AKdqqvA0LBhYQ-kjCthn-dVxf5VFZ9Yv?usp=sharing) afin de partager des documents et des images. Enfin, utilisons également des outils de discussion tels que Discord pour communiquer et rester en contact.



*Github Drive Discord*

# DICTIONNAIRES JAVA

La partie obligatoire du JAVA est 100% terminée. Toutes les commandes demandées sont implémentées. Le projet compte 4 classes et 3 fichiers de données afin de fonctionner. Tout est explicité en détail plus bas. Concernant le temps restant il sera consacré à finir la partie C à 100%. En fonction du temps que cela prendra, la partie optionnelle du sujet sera envisagée.

Le code JAVA est découpé en trois classes:

* **DictionaryMaker :** classe qui crée les différents dictionnaires ainsi qu’un fichier contenant les fréquences des lettres
* **DictionnarySearcher :** classe qui se base sur les fichiers créés par DictionnaryMaker pour trouver un mots et ses définitions
* **NormalizedExtractor :** classe qui renvoie tous les mots enregistrés dans les dictionnaires sous forme normalisée. Sa sortie peut être redirigé avec un ‘>’ afin de créer un fichier texte utilisé par la suite dans le C

Un visuel de cette architecture est disponible en [annexe](#_w4htj1yflc85).

Les trois fichiers générés par le classe *DictionaryMaker* sont :

* **XXX.json :** l’ensemble des mots et de leurs définitions sous format json
* **XXXsemi.lex :** l’ensemble des mots sous formes normalisées associées aux offsets de leurs versions non normalisées dans *XXX.json.* Offsets triées par ordre alphabétiques des mots auxquels il correspondent
* **XXX.lex :** l’ensembles des offsets des mots normalisés de *XXXsemi.lex*, triées par ordre alphabetique

\*XXX correspond au paramètre donné à *DictionaryMaker* comme nom des fichiers à générer.

Les commandes utilisables avec ces différents fichier :

* **java fr.uge.DictionaryMaker *path lang XXX* :** avec *path* qui indique le chemin absolu du fichier xml de génération, *lang* la langue voulue, *XXX* le nom des fichiers qui seront générés
* **java fr.uge.jdict.DictionarySearcher *XXX word* :** avec *XXX* le nom des fichiers dans lesquels il faut chercher et *word* le mot à rechercher. Le programme détecte automatiquement s'il est normalisé.
* **java fr.uge.jdict.NormalizedExtractor *XXX* :** avec *XXX* le nom des fichiers dans lesquels il faut chercher les statistiques. Renvoie dans la sortie courante tous les mots normalisés (peut être redirigé par ‘>’).
* **java fr.uge.jdict.CharTableMaker *XXX* :** avec *XXX* le nom des fichiers dans lesquels il faut chercher les statistiques. Renvoie dans la sortie courante chaques lettres et son nombre d’occurence dans les mots du dictionnaires (peut être redirigé par ‘>’).
* **java fr.uge.jdict.NormalizedExtractor *XXX "types, type, type"***. Permet d’extraire dans la sortie courante tous les mots normalisés appartenant aux types passés en paramètre (adjectif, verbe, nom ou nom propre)

\*XXX correspond au paramètre donné à *DictionaryMaker* comme nom des fichiers à générer

# MOTEUR DE JEU EN C

La partie C est presque terminée, d’après nous la partie C est réalisée à hauteur de 90% (il manque une seule fonctionnalité, mais qui n’est pas obligatoire en tant qu’alternant), de plus nous prenons en compte une petite marge afin de prévenir d’éventuels bugs ou problèmes d’optimisation que nous n’aurions pas encore trouver. Le moteur de jeu est actuellement découpé en 4 librairies et 6 fichiers exécutables (2 des librairies sont également des exécutables) soit un total de 8 fichiers.

Le code C est découpé en 4 librairies distinctes:

* **dictionnary.c** : la librairie qui regroupe les structures liées aux CSTree, StaticTree ainsi que toutes les fonctions utilisant uniquement le dictionnaire de mot. On y trouve des fonctions permettant de passer du fichier XXX.txt initial à un arbre, puis à un fichier XXX.lex représentant l’arbre (child-sisblings) du dictionnaire en .txt passé précédemment. Cette librairie contient également la fonction dictionnary\_lookup permettant la recherche d’un mot dans le dictionnaire pour vérifier s’il est valide.
* **grid.c** : la librairie qui permet de gérer les grilles, avec la structure utilisée, les méthodes pour passer de grille “2D” à “3D” et inversement. On y trouve également les méthodes permettant de créer des grilles aléatoires en utilisant les fréquences des lettres dans le dictionnaire. Ainsi que les fonctions permettant de vérifier si un mot est valide dans une grille.
* **solve.c** : (actuellement en cours de travail) la libraire qui permettra de retourner tous les mots présents dans une grille qui sont également valide dans le dictionnaire indiqué par l’utilisateur
* **score\_by\_length.c** : qui contient la méthode permettant de calculer le score en fonction de la longueur du mot entré par le joueur.

Il y a ensuite 8 fichiers pouvant être exécutées :

* **dictionnary\_build** : pour créer un dictionnaire en .lex qui est utilisé dans tout le reste du moteur de jeu. Il suffit pour cela de lui transmettre le nom du fichier utilisé et le nom du fichier en sortie.
* **dictionnary\_lookup** : pour vérifier si un mot est valide dans le dictionnaire. Il faut lui transmettre le fichier .lex du dictionnaire utilisé ainsi que le mot recherché, et le programme renvoie une valeur numérique. 0 le mot est valide, 1 le mot est préfixe d’un mot valide et 2 le mot n’est pas valide.
* **grid\_build**: qui créer une grille aux dimensions indiquées avec les fréquences souhaitées. Il faut pour cela lui transmettre le fichier de fréquence et les dimensions.
* **grid\_path**: pour vérifier si un mot est valide dans une grille, il faut transmettre le mot recherché, les dimensions de la grille et la grille. Le fichier renvoie 0 si le mot est trouvé et 1 sinon.
* **solve** : qui prend en paramètre le fichier .lex du dictionnaire souhaité, la longueur de mot souhaité, les dimensions de la grille et la grille pour renvoyer la liste des mots valide dans le dictionnaire, dans la grille et ayant une longueur suffisante.
* **score\_by\_ length**: auquel il faut transmettre une liste de mots, et qui renvoie le nombre de points marqués par cette liste de mots.

# 

# BASE DE DONNÉE

Comme dit dans l’introduction la base de donnée n’est que légèrement avancée en raison de l’arrivée tardive du sujet. Nous avons donc fini d’estimer les besoins. Il faut maintenant avancer sur l’écriture des schéma, dictionnaire et mise en place de la base de données.

Pour rentrer dans les détails des besoins identifiés. Il faut que nous puissions stocker/retrouver différentes informations sur les parties de boggle, nous avons identifié différentes informations nécessaires : l’id de la partie, le dictionnaire utilisé, la grille utilisée et la date à laquelle la partie à été jouée.

Le même besoin s’applique aussi au joueur avec : son identifiant, mail, mot de passe ,pseudonyme, description personnelle, photo de profil et date de création du profil. En plus nous pensons pouvoir récupérer la date de la dernière partie jouée ainsi que le nombre de parties jouées grâce à des requêtes SQL. Pas besoin de les stocker donc.

En plus de cela nous avons besoin de savoir quel joueur a joué quelle partie. Connaître le score de chaque joueur ainsi que son temps moyen pour trouver un mot pour une partie donnée. Toujours sur une partie nous aimerions connaître tous les mots joués par un joueur, en différenciant les mots valides/invalides.

Enfin, pour les besoins, nous aimerions que chaque joueur puisse décider si son profil est public ou privé. Un champ indiquant cette information semble donc nécessaire.

Suite à cela nous avons repris notre premier schéma pour l’adapter à nos besoins. Vous pouvez les consulter dans l’[annexe.](#_kekd9myjsuas)

Concernant le SGBD nous avons fait le choix d’utiliser MySQL pour son interface graphique intégré dans les serveurs de l’iut. Interface qui permet de consulter facilement et clairement les tables, sans doute de grande taille pour les différents historiques, de notre base de données.

# PROBLÈMES RENCONTRÉS

Au niveau du java les problèmes rencontrés ne sont pas nombreux. La partie la plus dure étant de penser la structure des différents fichiers de stockages de sorte à ce que les différentes commandes ne soient pas trop gourmandes en temps de calcul. D'où cette architecture en quatres fichiers qui prend un certain temps à ce générer mais améliore grandement la vitesse de recherche par la suite.

Au niveau du C, la mise en place de l’architecture fût laborieuse. D'autre part, nous avons eu beaucoup de soucis liés aux fuites mémoires qui rendaient difficile et plus longue la réalisation de cette partie.

# FUTUR

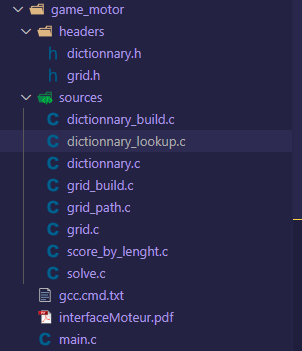
Nous envisageons d’utiliser un outil proposé par M. *CHILOWICZ* qui est Valgrind afin de trouver la source de nos fuites mémoires dans le moteur de jeu en C.

Concernant la base de données, nous prévoyons de normaliser notre schéma afin d’éviter toute dépendance fonctionnelle. Une fois cela fait nous devrons rédiger le dictionnaire de données de la base. Pour finir, une fois que la base sera mise en place, nous créerons le jeu de requête. A noter que cette partie attendra des éclaircissements concernant la partie web.

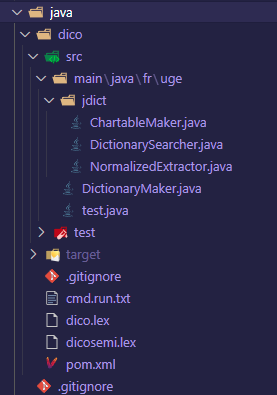
Enfin, en fonction des avancées de la partie C et de la base de données, envisager la mise en place de la compression des données en JAVA.

# 

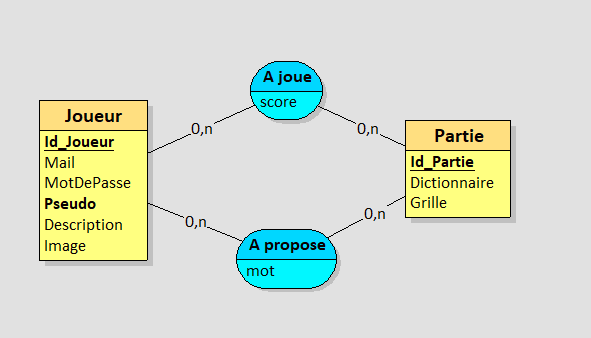
# ANNEXES



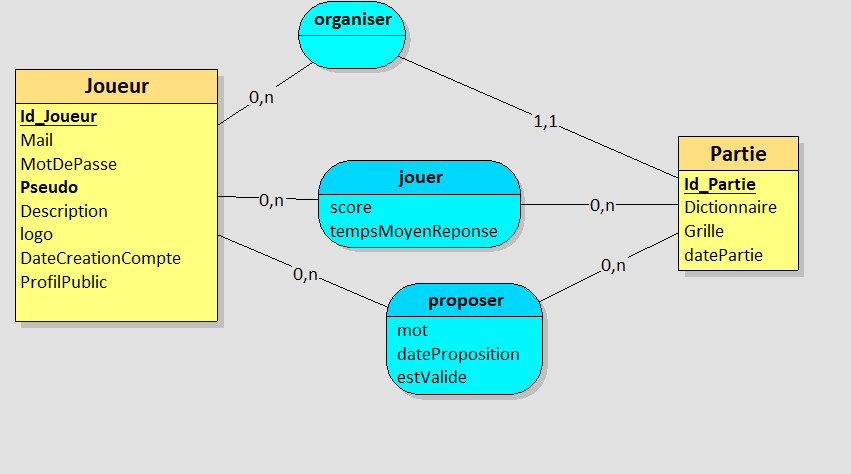
***hiérarchie de la partie C***



***hiérarchie de la partie JAVA***

******

***Premier schéma de la base de donnée***

******

***Second schéma de la base de donnée***