Mardi Apres midi : Après la présentation du projet on a fait une simulation client ou chacun d’entre nous s’est mis dans la peau du client et du développeur, et le client exposait ce qu’il attendait exactement. Pour donner suite à cela on a établi un schéma textuel pour le cheminement du projet afin d’avoir une démarche claire.

On a tout d’abord effectué une recherche du jeu en téléchargeant l’application pour que cela nous aide à mieux comprendre les étapes et savoir comment ce dernier marchait.

Ensuite on a établi le besoin : le système de jeu, les différentes cartes, le paiement, les abonnements, la partie sociale, les utilisateurs...

On a séparé les différents besoins dans différentes sections de technologies adaptées.

Tout ce qui va être partie sociale va être SQL

API tiers pour tout ce qui est abonnement, paiement...

Le système de jeu, les différentes cartes... ça va être notre partie back

Ensuite on a utilisé la méthode du Behavior Driven Development en faisant un scénario du déroulement d’une partie de jeu ; et voici le processus que nous avons élaboré :

# Partie

## Etape 1 (Recherche de partie)

Lorsque le joueur cherche une partie, il est mis dans un groupe par rapport à son score. On essaie de regrouper 8 joueurs avec un score dans les mêmes tranches (plus le temps passe plus la tranche augmente). Puis dès que les 8 joueurs sont trouvés on leur attribut un serveur auquel ils se connecteront.

## Etape 2 (Définition des types de carte)

On fait une sélection aléatoire de 4 des 8 types de carte qui existe.

## Etape 3 (Sélection du Champion)

On dispose de 4 champions par joueur, chaque Champion apparait uniquement pour 1 seul joueur (on doit créer un minium de 4\*8 champions pour le lancement). Si le joueur ne possède pas d’abonnement on va lui retirer la possibilité de choisir 2 des 4 champion. Chaque champion possède une compétence (Passive ou Active) et 40 points de vie au début de la partie.

## Etape 4 (Combat)

Cette phase se sépare en 2 phases, Le première est la phase de préparation et l’autre celle de combat. On alterne entre les 2 jusqu’à ce qu’il ne reste plus qu’un joueur en vie.

## Etape 5 (Préparation)

Cette phase est le seul moment où le joueur peut interagir avec le jeu. Il a la possibilité d’acheter des créatures, les vendre et les placer sur son « board ». En plus de ça, il a la possibilité d’augmenter son niveau de taverne, changer la disponibilité des créatures de la taverne ou de la bloquer pour acheter au prochain tour. Il a aussi la possibilité d’utiliser son pouvoir de champion pendant cette phase.

## Etape 6 (Battle)

Lors de cette phase, le joueur se retrouve en combat avec un autre joueur aléatoirement. 1 des 2 sera choisi au hasard pour commencer à attaquer. C’est le monstre le plus à gauche qui n’a pas attaqué qui va mener l’attaque, puis la créature la plus à gauche de l’autre joueur va attaquer, ainsi de suite jusqu’à ce qu’un joueur ne possède plus de créature. Si toutes les créatures d’un joueur ont attaqué, on recommence au début de la file. Quand 1 des joueurs n’a plus de créature sur son « board » on vérifie si l’autre joueur en possède au moins. S'il en possède, le joueur n’en possédant plus perd des points de vie correspondant au niveau du joueur qui en possède plus le niveau de créature toujours ne vie après le combat. Si aucun des 2 joueurs ne possède de créature, personne de perd de point de vie. Quand 1 joueur n’a plus de point de vie, il est éliminé et sa position sera fixe lors de la suppression ou l’ajout de score.

## Fin du combat

On regarde la position de chaque joueur et on leur attribut des récompenses en fonction de leur place. En plus de cela, les 4 premiers joueurs gagneront du score et les 4 derniers en perdront.

Mercredi :

Après avoir textuellement écrit tout ce dont nous devrions avoir besoin pour la satisfaction du client dans ce projet, on l’a également matérialisé sous forme de schéma qui trace globalement le déroulé du jeu. C’est à dire la création d'un diagramme de classe ou on met toutes les étapes du jeu :

Une image contenant texte, diagramme, Plan, Dessin technique

Description générée automatiquement

# Créature

Vu que les créatures sont les pièces maitresses du jeu, nous avons listé tout ce dont on a besoin pour les créatures :

Chaque créature possède plusieurs valeurs :

* Point de vie : Point qui correspond à la vie de la créature, elle tomber a 0 ou moins, la créature meurt pour le combat.
* Attaque : Point qui correspond au dégât fait à la créature adverse quand il attaque.
* Type : Type de la créature, peut influencer sur certain effet
* Effet : capacité particulaire d’une créature. Il existe verbe d’effet qui déclenche plusieurs effets.
  + Râle d’agonie : Déclenche l’effet après la mort de la créature
  + Cri de guerre : Déclenche l’effet lorsque la créature est posée sur le « board »
  + Bouclier divin : La première attaque sur la créature ne lui fait pas de dégât.
  + Provocation : La créature sera ciblée en priorité par les attaques des créatures adversaire.
  + Furie des vents : LA créature peut attaquer 2 fois
  + Camouflé : La créature est inciblable temps qu’elle n’a pas attaqué
  + Sorcellerie : Donne un sort à chaque début de la préparation
* Levels : Défini le niveau du joueur pour avoir une chance d’avoir la carte dans la taverne.

## Déclinaison des créatures

Afin de créer des instances de créature, nous utiliserons une class mère Créature pour la faire hériter à chaque type de créature. Ce qui voudra dire que nous aurons 1 classe par créature existante. Chacune de ces classes possèderont les fonctions de déclanchement de passif tels que Rale d’agonie, crie de guerre, etc…

Les technologies que nous avons choisi d’utiliser sont :

# Technologies :

## Front : C# & Unity

* Compatibilité entre Android et IOS
* Capacités avancées en matière de graphismes et d'animations
* Moins gourmand que Unreal Engine
* Moteur de jeux

## Back : MLAPI & NestJS

* Plugin Unity
* Plugin développer par Unity Technologies
* Très populaire.
* Ressemblance a du Web Socket.
* Possibilité de créer des serveurs dédier pour rediriger les utilisateurs vers le serveur le plus proche d’eux.
* Permet de gérer facilement les connexions des utilisateurs.
* Construction par Objet
* Facile d’utilisation / Apprentissage

## BDD : Firebase & PostgresSQL

* Gestion d'authentification
* Solution tout en 1 (Model, Son, Donné, User)
* Mailing et SMS Manageur intégrer
* Diffèrent localisation des serveurs pour temps des requêtes plus courte selon le serveur dédier sélectionner
* Possibilité de clustering
* Firestore pour la modularité
* SQL pour la partie Boutique

# Info Back

## MLAPI

MLAPI est une bibliothèque open-source développée par Unity Technologies pour faciliter le développement de jeux multijoueur en réseau avec Unity. Dans notre architecture, MLAPI sera utilisé comme solution back-end en partie pour gérer les connexions des utilisateurs, la synchronisation de l'état du jeu et les interactions en temps réel entre les joueurs.

On l’utilisera aussi pour créer des serveurs dédiés qui redirigent les utilisateurs vers le serveur le plus proche d'eux, ce qui améliore la latence et l'expérience utilisateur.

## API REST

En plus du MLAPI, nous créerons une API en NestJS, cela nous permettra de créer des interactions entre les joueurs lorsqu’ils ne sont pas en partie. Cela pourra être utilisé pour récupérer des informations sur la boutique, sur les joueurs et même sur leurs collections. En plus des requêtes http, nous mettrons en place un service de web socket afin que les utilisateurs puissent recevoir des données en temps réel tels que des messages privés des nouveaux objets de la boutique, etc…

# Base de Donnée

## Firestore

C’est la base de données de documents utilisés pour stocker des informations sur les parties de jeux. Celle-ci sera utile pour la facilité de modification de la donnée stockée par rapport à l’état de la partie ou celle des créatures de la partie.

Cette base de données aura une disponibilité de 99.999% ce qui sera utile car on doit pouvoir écrire sur la base de données à tout moment et avec la plus grande rapidité car le temps d’action du joueur est restreint dans le temps à cause d’un timer.

En plus de cela, la base de données sera résiliente pour éviter de supprimer les parties récentes si la base de données crash. Nous pourrons aussi dupliquer la structure des documents sur d’autre base afin de réduire le temps de réponse entre les diffèrent serveur de jeux qui seront aussi déployer dans tout le monde pour réduire au maximum le ping des joueurs.

## Realtime DataBase

Cette base de données servira à stoker les informations des joueurs tels que son pseudo, score, ses collection, etc… Pour se faire nous stockerons toute ces informations dans une seul et même table utilisateur qui stockera tous les id de ses collections. Nous ne stockerons pas ces informations dans le Firestore pour réduire les coups de Firebase.

## Postgres SQL

Cette base de données stockera les éléments de la boutique du jeu, cela nous permettra de structurer la donnée et de l’uniformiser, en plus de cela la base de données SQL sera moins couteuse que celle de Firestore.

## Storage

Cette base de données sera utilisée pour stocker les modèles et le son du jeux pour que quand l’utilisateur téléchargera le jeu, celui-ci téléchargera les model depuis le Storage de Firebase. En plus de cela, les image de profil de chaque joueur pourront être stocker sur cette base de données.

# Service Tiers

## Authentification

En plus des bases de données présentées au-dessus, nous utiliserons la fonctionnalité Authentification de Firebase afin de déléguer le système de connexion à Firebase et aussi afin de gérer les connexions par SSO.

## Stripe

Afin de gérer la formule économique du jeu, nous vendrons des skins sur la boutique du jeu et aussi la possibilité de souscrire à un abonnement afin de débloquer des fonctionnalités en plus et des récompenses. Pour cela un utiliseront l’API tiers de strippe, celle-ci nous permet d’avoir plusieurs moyens de paiement (visa, carte bleu, PayPal, etc…) en plus de cela Strippe possède une fonctionnalité d’abonnement performent ce qui nous soulage du développement de la solution afin de gérer l’abonnement du joueur au jeu. Strippe permettra aussi de gérer la facturation de nos client (la création des factures après achat sur la boutique).

# Estimation de budget

## Hébergement

### MLAPI

Pour héberger nos serveurs de jeu MLAPI, nous utiliserons des serveur t3.xlarge qui nous donne 4 vCPU, avec 16 GiO de RAM. On estime qu’un utilisateur utilisera 0.16GiO de RAM. Ce qui nous donner 100 joueurs par serveur. Si 1 serveur coute 86,51 dollars par mois, nous auront besoin de (1.000.000/100) \* 86,51 = 800.000 par mois. Il faudrait repartir les diffèrent serveur sur le globe afin d’avoir des temps de réponse acceptable sur toute la planète.

### API

N'ayant pas besoin d’une infrastructure aussi importante que pour le MLAPI, l’API en Nest aura besoin de beaucoup moins de puissance.

### Firebase

On va utiliser 1 To pour stocker les donné utilisateur (real time). Afin de gérer les informations stocker lors des parties, nous opterons pour un capaciter de stockage de 500 GiO (firestore) repartie sur diffèrent continent pour les relier au serveur le plus proche. Pour le SQL, nous opterons pour une base de données 1To. Et enfin nous finirons par le Storage pour stocker les diffèrent fichier, pour cette base de données, nous prendrons aussi 1To afin de ne pas être limiter par les photos de profil des utilisateurs.

Celons les prix de Firebase, nous aurons 580$ + 350$ + 25$ (pour firebase) et 70$ pour le PostgresSQL.

Bien évidemment, nous dupliqueront toutes ces bases de données afin de créer de la récursivité et d’avoir des backups.

## Développeur

Le projet se développera sur 2 ans, nous embaucherons 2 devs Unity, 2 devs MLAPI, 1 dev NestJS, 2 Designeurs, 1 Sound Designeur :

* Dev Unity : 100 000$ / ans
* Designeur : 90 000$ / ans
* Designeur 3D : 100 000 / ans
* Sound Designeur : 100 000 / ans
* Dev MLAPI : 120 000 / ans
* Dev NestJS : 90 000 / ans

Pour un total de 920 000 / ans soit un budget de développement de 1,84 millions de dollar.

# C2

# Gestion du projet

## Versionning

Pour gérer le versionnement de notre projet de manière efficace, nous opterons pour l'utilisation de GitLab, une plateforme de gestion de code source et de collaboration. Grâce à GitLab, nous pourrons implémenter un pipeline CI/CD (Continuous Integration/Continuous Deployment) afin d'automatiser le processus de construction et de déploiement de notre application sur nos environnements de test et de production.

Pour maximiser la portabilité et la gestion des environnements, nous intégrerons Docker à notre infrastructure. En utilisant des conteneurs Docker, nous pourrons encapsuler notre application et ses dépendances dans des unités autonomes et reproductibles. Cela nous permettra de garantir une cohérence entre nos différents serveurs de test et de production, tout en facilitant le déploiement et la mise à l'échelle de notre application. En combinant GitLab et Docker, nous créerons un pipeline CI/CD robuste et flexible, capable de garantir une livraison continue et fiable de notre projet.

Pour protéger notre environnement de production, nous avons décidé de restreindre les pushs de commit directement sur la branche production. La seule méthode autorisée pour intégrer des modifications dans cette branche sera de créer une pull request depuis notre branche de développement. Avant de valider cette pull request, nous mènerons une série de tests intensifs sur un environnement de pré-production pendant une semaine. Cette période de test nous permettra de vérifier la fonctionnalité et la fiabilité de la nouvelle version, garantissant ainsi une transition fluide vers la production.

## Methode

Dans notre méthode de développement, nous avons choisi d'adopter une approche centrée sur le Test-Driven Development (TDD). Cette approche nous permet de garantir la qualité du code dès le début du processus de développement en nous concentrant sur les exigences fonctionnelles et les comportements attendus du système. En écrivant les tests en premier, nous clarifions nos objectifs de développement et définissons les critères de réussite pour chaque fonctionnalité. De plus, les tests automatisés agissent comme une spécification vivante du système, fournissant une documentation précise et à jour de son comportement attendu. En plus de ça, nous nous engageons à respecter les principes SOLID, une série de principes de conception orientée objet qui favorisent la création de code flexible, maintenable et extensible.

Dans notre projet, nous attachons une importance particulière à la qualité du code, et c'est pourquoi nous intégrerons SonarQube dans notre processus de développement. SonarQube est un outil d'analyse statique de code qui identifie les problèmes potentiels, les violations des bonnes pratiques de programmation, les vulnérabilités de sécurité et les inefficacités de performance dans notre code source. En utilisant SonarQube, nous pouvons automatiser l'évaluation de la qualité du code à chaque étape du développement, en nous assurant que notre code respecte les normes de programmation établies, telles que la cohérence du formatage, la gestion des erreurs, la complexité cyclomatique, etc. De plus, SonarQube fournit des rapports détaillés et des métriques sur la qualité du code, ce qui nous permet de suivre les progrès au fil du temps et d'identifier les zones à risque ou nécessitant une amélioration.