Sorbonne Université & CFA INSTA

05 novembre 2023

Rapport projet 2 DAAR

Jeu de carte à collectionner

Auteurs :
Raphaël Françoise
Farouck Cherfi

Encadrant :
Guillaume Hivert
Binh-Minh Bui-Xuan

Table des matières

1 Introduction

Dans le cadre de l'unité d'enseignement Développement des Algorithmes d'Application Réticulaire, nous nous sommes attelés à développer une application décentralisée basée sur la blockchain Etherum. L'objectif étant de proposer aux utilisateurs un jeu de carte à collectionner. Ainsi, nous avons créé des collections de cartes représentées sous forme de NFT, une marketplace d'échange de NFT entre utilisateurs, un système d'ouverture de booster aléatoire et la possibilité pour un utilisateur de visualiser tout son wallet de NFT depuis l'application.
Dans ce rapport, nous allons vous présenter la structure de notre application ainsi que son fonctionnement. Nous aborderons aussi les difficultés rencontrées lors de la réalisation de notre projet.

2 API & Base de donnée

Concernant le jeu de carte, nous avons décidés d'implémenter le grand classique qu'est Pokémon TCG [8]. Afin de récupérer tous les set (dans ce qui suit nous utiliserons set et collection afin de décrire la même chose, un ensemble de cartes parues lors de la même édition) et les cartes existantes nous avons utilisés l'API Pokemon TCG [7]. Discutons maintenant plus en détail de comment avons nous récupérés ces données.

Tout d'abord, toutes les données en rapport avec les collections et les cartes sont stockées en dur sous forme de fichiers **JSON** dans le fichier **db**. Afin de récupérer ces données nous avons utilisés un script python `apiTCG.py` présent dans ce même fichier.
Abordons désormais les fichiers collectés ainsi que leur composition. Nous avons dans un premier temps récupéré toutes les collections de carte existantes dans le fichier **collections.json**. Pour chaque collections les champs qui nous intéressent sont :

- **id** : l'id unique de la collection
- **name** : le nom de la collection
- **printedTotal** : le nombre de cartes différentes dans la collection
- **images** : permet d'accéder aux urls du symbole ou du logo de la collection

Ensuite, pour chaque collections, nous avons récupéré toutes les informations concernant les cartes des collection. Nous avons donc autant de fichier JSON que de collections. La nomenclature des fichiers étant `id de la collection + "-cards.json"`. Chaque fichier est donc composé de toutes les informations concernant les cartes de la collection. Pour chaque carte nous récupérons les information suivantes :

- **id** : l'id unique de la carte
- **name** : le nom du pokémon
- **images** : les urls des cartes en différentes tailles

Toutes ces données nous sont utile lorsque nous voulons afficher les cartes dans le client, créer les collections dans la blockchain et miner les NFT de chaque cartes.

3 Blockchain

Nous n'avons pas ajouté de contrat en plus du contrat Collection.sol et Main.sol. Lors du déploiement de la blockchain, nous avons choisi de déployer uniquement le contrat Main, qui jouera un rôle de façade pour les contrats de collections. Au lancement de la blockchain, nous avons créé les 156 contrats de collections correspondant aux collections existantes de l'API TCG.

3.1 Fonctionnalités

3.1.1 Création de collections

Pour commencer, nous avons une fonctionnalité qui permet de créer une collection avec un nom et un nombre de cartes représentant le nombre total de cartes dans cette collection, par exemple 151 cartes qui peuvent exister en plusieurs exemplaires

3.1.2 Récupération des collections

Une fonctionnalité en lien avec la création de collections qui permet de récupérer tous les noms des collections de la blockchain, ce qui permet ensuite d'effectuer des opérations dessus

3.1.3 Minter des cartes

La fonctionnalité de mint nous a permis de créer des boosters. Grâce à celle-ci, nous pouvons choisir la collection dans laquelle nous voulons mint une carte, indiquer l'adresse du futur propriétaire du NFT, ainsi que des URI composés de l'ID de la collection + le numéro de la carte. Cette fonctionnalité permet alors d'utiliser la méthode safeMint de la collection, qui permet de créer un NFT et de générer un tokenID unique pour celui-ci.

3.1.4 Possessions des NFT

Nous avons deux fonctionnalités : l'une permettant de savoir combien de NFT l'utilisateur possède dans une collection, et l'autre permettant de récupérer tous les NFT de cette collection. Nous récupérerons alors les URI ainsi que les tokens pour faciliter les transferts et les achats de ces NFT

3.1.5 Achats de NFT

Cette fonctionnalité permet d'acheter un NFT qui a été mis en vente sur la marketplace d'une collection en particulier. L'acheteur récupère alors le tokenID et le nom de la collection pour appeler cette fonction. Elle utilise la méthode d'achat de la collection, qui effectue un transfert de NFT du vendeur à l'acheteur pour ce tokenID.

3.1.6 Ventes de NFT

La vente de NFT nécessite l'adresse du vendeur, le nom de la collection ainsi que le tokenID de la carte. Comme nous récupérons ces informations depuis la fonctionnalité de possession des NFT, il suffit alors d'appeler cette méthode avec les paramètres nécessaires. Le NFT appartient toujours à son vendeur jusqu'à son achat. Pour représenter une vente, nous avons utilisé une structure appelée Vente qui permet de stocker toutes les informations nécessaires pour l'achat, telles que l'adresse du propriétaire actuel, le prix (même si nous ne pouvons pas modifier ou envoyer d'argent), l'URI ainsi que le tokenID. Nous demandons généralement l'adresse du vendeur ou de l'acheteur car l'adresse qui appelle les contrats est l'adresse du contrat principal. Nous devons donc récupérer cette information pour la plupart des fonctionnalités utilisant les collections.

3.1.7 MarketPlace

Nous avons la possibilité de récupérer tous les NFT qui ont été mis en vente par des utilisateurs grâce à cette fonctionnalité. Elle récupère tous les tokenID, les URI ainsi que les prix de chaque carte.

4 Serveur

Pour ce qui est du backend de notre application, nous avons mis en place un serveur afin de gérer différentes requêtes utilisateur. Pour mettre en place ce serveur, nous utilisons **Node JS** [5] et plus particulièrement le framework **Express JS** [2] et **Axios** [1] (client HTTP). De plus, le code concernant le serveur se situe dans le fichier `backend/server.js`. La principale utilité de ce serveur est de récupérer des informations située dans notre base de donnée. En effet stocker des informations directement dans la blockchain est très coûteux, il est donc nécessaire de pouvoir y accéder depuis un autre endroit.
Dans cette partie nous allons principalement présenter les différentes requêtes mise en place et leur utilité au sein de notre application décentralisée.

Développons les différentes requêtes en fonction de leur route :
- Route `"/collection"` [GET] : l'appel à cette route avec la méthode get permet de récupérer les images et le nom de toutes les cartes de la collection possédant l'id `id`. Pour cela nous récupérons dans l'id de la collection en temps que query de la requête. Cette requête est effectuée lorsque l'on clique sur une collection depuis la page de `collections`.
- Route `"/collectionsData"` [GET] : l'appel à cette route avec la méthode get permet de récupérer l'ensemble des informations (nom, image, symbole, série, ...) de toute les collections passées en query de la requête. Cette requête est effectuée dans la page de `collections` et la page d'**achat** afin d'afficher les informations des collections dans ces pages.
- Route `"/booster"` [GET] : l'appel à cette route permet récupérer de manière pseudo-aléatoire (module random javascript) les informations de 5 cartes d'une collection passée en query de la requête. Cette route correspond à l'ouverture aléatoire d'un booster. Il est donc important de crypter la réponse. Pour cela nous utilisons `crypto-js`. Cette requête est donc effectuée dans la page d'`achat` lorsque nous ouvrons un booster.
- Route `"/id"` [GET] : l'appel à cette route permet de récupérer toutes les informations concernant les cartes d'un joueur passées en query de la requête.

5 Client

Pour la partie frontend de l'application, nous avons repris le squelette mis à disposition sur le repo github [3]. Nous avons donc utilisé **ReactJS**. Afin d'utiliser la puissance de React, nous avons mis en place un fichier `components` qui des composants que nous réutilisons plusieurs fois dans le code, tel que le composant `Cards`. De plus, le fichier `pages` contient toutes les différentes "pages" de notre application tels que `Mes cartes`, `Marché`, ...
Le wallet de l'utilisateur est donc géré grâce à la fonction `useWallet()`. Les différents appels aux fonctions de la blockchain se font via ce wallet en appelant le contrat `Main`.

Dans le front nous avons donc accès à toutes les fonctionnalités de l'application.
- `Acceuil` est la page sur laquelle on arrive en démarrant l'application.
- `Collections` est la page qui contient toutes les collections disponibles dans la blockchain
- `Mes cartes` est la page qui affiche les cartes de l'utilisateur connecté via son wallet metamask [4]
- `Achat` est la page au sein de laquelle on peut ouvrir un booster.

— `Marché` est la page au sein de laquelle on accède à la marketplace commune des utilisateurs.

Désormais nous allons donner une petite explication des pages les plus importante de l'application.

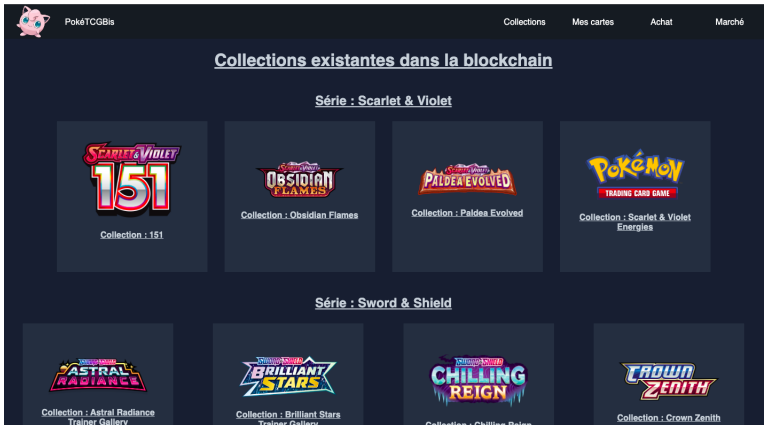

FIGURE 1 – Page collection

Ci dessus, nous pouvons voir la page collection. Au sein de cette page, lorsque nous cliquons sur une collection, nous avons accès à toutes les cartes de la collection.

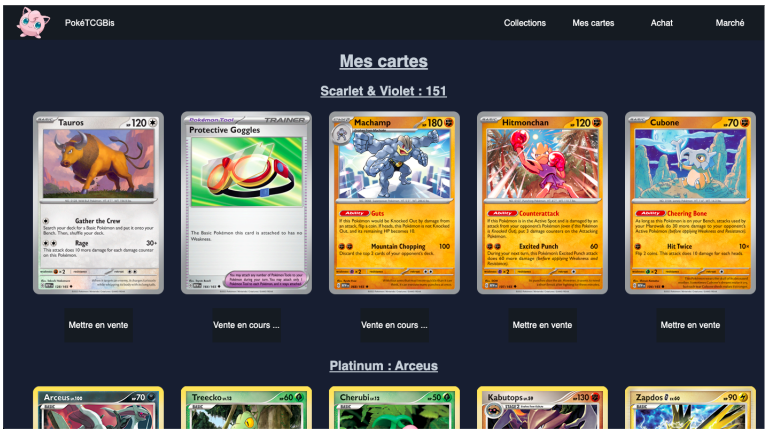

FIGURE 2 – Page mes cartes

Ci dessus, la page **Mes cartes** à l'utilisateur d'accéder à toutes ses cartes. Il peut décider de les vendre en cliquant sur le bouton `Mettre en vente`. Une fois cela fait, il est indiqué que la carte est en vente avec `Vente en cours ...`

Ci dessus, la page achat permet de sélectionner une collection et d'ouvrir un booster de cette dernière en cliquant sur le bouton `Ouvrir un booster`.

Ci dessus, la page marché au sein de laquelle les utilisateurs peuvent acceder aux cartes en vente et en acheter en cliquant sur la bouton `Acheter`.

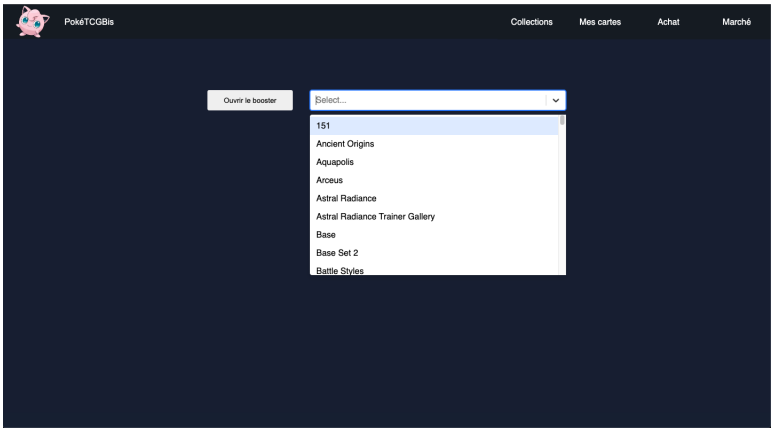

FIGURE 3 – Page achat

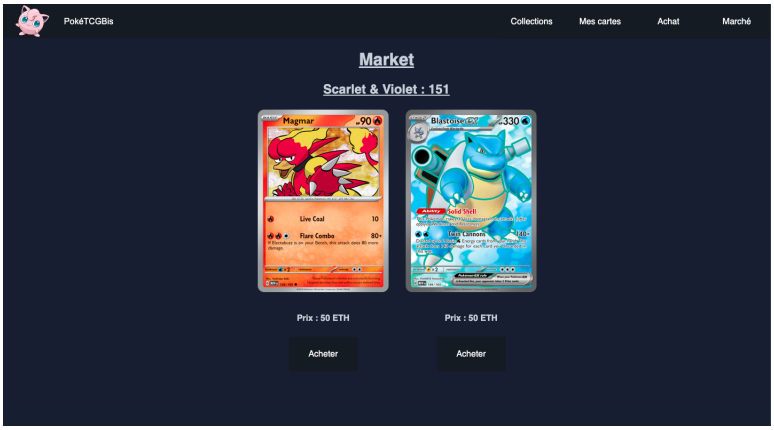

FIGURE 4 – Page marché

6 Difficultés rencontrées

6.1 Installation

Nous avons eu beaucoup de mal à installer hardhat ainsi que Oppenzeppelin [6] en raisons de ces dépendances et ces versions.

6.2 Premier obstacle : Création des collections

Nous avons longuement hésité sur la décision de déployer les collections une seule fois au démarrage, ou de les déployer lorsque qu'un utilisateur voulait accéder à une collection. Nous avons beaucoup réfléchi pour déterminer si le déploiement des contrats de collection devait se faire depuis le frontend, lors du déploiement de la blockchain ou depuis le backend

6.3 Connexion de la blockchain avec le backend

Nous n'avons pas réussi à connecter la blockchain au backend en raison des signers et des providers avec Web3. Nous avons alors décidé d'utiliser Ethers pour le backend. Cependant, lorsque nous déployions le contrat depuis l'arrière-plan, une nouvelle instance du contrat Main était créée

car l'adresse du contrat ABI Main n'était pas la même. Nous avons donc choisi de ne pas établir de connexion entre le backend et la blockchain, malgré son efficacité pour stocker les informations de la blockchain lorsqu'il y a un changement, et d'appeler uniquement le backend pour récupérer les informations au lieu de faire des appels excessifs vers la blockchain

6.4 Aléatoire du booster

Nous avons décidé d'opter pour un choix pseudo-aléatoire depuis le backend. Nous avons crypté la réponse du backend vers le frontend afin que l'utilisateur ne puisse pas accéder aux informations des cartes. Pour éviter une attente trop longue lors du mint des cartes lorsqu'un utilisateur ouvre un paquet de cartes, nous avons décidé d'afficher les cartes tirées et de minter les cartes en parallèle. Comme l'utilisateur signe la transaction sur MetaMask, il ne peut pas annuler l'écriture dans la blockchain.

Nous voulions également ajouter un système d'aléatoire basé sur la rareté, c'est-à-dire obtenir un booster de 10 cartes où il y a 7 cartes communes, 1 énergie, 1 SuperRare et une UltraRare. Cependant, faute de temps, cette fonctionnalité n'a pas été ajoutée. Pour ce faire, nous aurions pu vérifier la collection et récupérer la rareté de chaque carte, puis effectuer notre aléatoire sur un nombre précis de cartes communes, etc..

6.5 MarketPlace

Nous avons réfléchi à un système d'échange de cartes entre utilisateurs et à un système de vente avec la possibilité de modifier les prix. Malgré quelques essais et recherches, nous n'avons pas réussi à mettre en place le transfert d'argent fictif d'un compte à un autre. En ce qui concerne l'échange de cartes, nous souhaitions laisser aux utilisateurs la liberté de choisir les cartes avec lesquelles ils étaient prêts à effectuer l'échange, afin de simplifier le processus

6.6 Collection

Notre première approche pour l'aperçu des cartes d'une collection consistait à les assombrir pour montrer à l'utilisateur qu'il ne les possédait pas. Une fois un booster ouvert et des cartes en sa possession, les cartes étaient censées s'illuminer pour informer l'utilisateur de sa possession de ces cartes et il pouvait alors les vendre depuis l'onglet Collections.

7 Conclusion

Nous avons énormément apprécié ce projet, car il nous a permis de mieux comprendre le fonctionnement de la blockchain et le développement qui peut exister derrière. Entre traiter des données dans la blockchain, que tout le monde peut lire, entraîne une augmentation du coût en gas et de l'autre, les traiter dans le backend permet de cacher et d'empêcher à l'utilisateur de voir ce qu'il se passe, tout en économisant les coûts en gas de la blockchain. C'est un compromis auquel nous avons dû faire face pendant ces quatre semaines.

Références

[1] Axios. https://axios-http.com/fr/docs/intro.

[2] ExpressJS. https://expressjs.com/fr/.

[3] Guillaume Hivert. Github du projet. https://github.com/ghivert/collectible-card-game-daar.

[4] MetaMask. https ://metamask.io/.

[5] NodeJS. https://nodejs.org/en.

[6] OppenZeppelin. OppenZeppelin. https ://www.openzeppelin.com/.

[7] pokemontcg.io. PokemonAPI. https://pokemontcg.io/.

[8] Wikipedia. PokemonTCG. https://en.wikipedia.org/wiki/Pok%C3%A9mon_Trading_Card_Game.