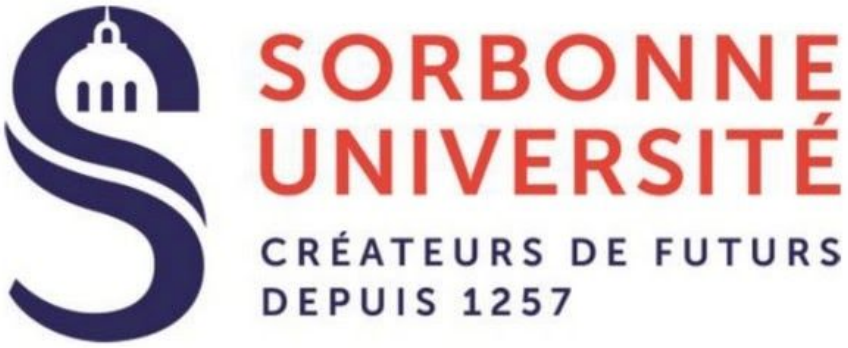

Rapport de projet

Decentralized Wikipedia

Auteurs: Hakim CHERBAL
Skander SERSI
Mohamed NIZAMUDDIN

Introduction

Le projet a pour but le développement d'une première application décentralisée (DApp) sur la blockchain Ethereum, en utilisant le langage Solidity et le framework Truffle, l'application en question sera un site décentralisé de publication d'articles comme wikipédia, il sera basé sur l'utilisation de la blockchain pour le stockage des articles.
L'application sera divisée en deux grandes parties, la partie front-end sera en React Js, la partie back-end sera en Ethereum.

1 . Le contrat intelligent

1.1. Représentation des articles dans le smart contract

```
4   struct Article {
5     string content;
6   }
7
8   uint[] public ids;
9   mapping (uint => Article) public articlesById;
10
11  //Tableau permettant de récupérer le dernier identifiant de l'historique d'un article
12  mapping (uint => uint) public idsOfHistory;
```

Figure 1: La structure de donnée représentant les articles dans le contrat

Un article sera représenté à l'aide d'une struct qui recevra le contenu de l'article dans un string (ligne 4-6).
Les articles seront contenus dans une structure de donnée de type map, qui aura en clé l'index d'un article et en valeur l'article en lui-même (ligne 9).
Les index seront eux aussi contenus dans un tableau d'index (ligne 8).

1.2. Représentation de l'historique des articles

```
11  //Tableau permettant de récupérer le dernier identifiant de l'historique d'un article
12  mapping (uint => uint) public idsOfHistory;
13
14  //Tableau contenant l'historique de modification d'un artcile
15  mapping (uint => mapping(uint => Article)) public historyOfArticles;
```

Figure 2: La structure de donnée représentant l'historique des articles

L'historique est représenté par deux map, un contenant l'index de la dernière modification pour chaque article (ligne 12), et l'autre map va contenir en clé l'index d'un article, et en valeur un map contenant tous l'historique de l'article auquel la clé fait référence, chaque historique sera ainsi représenté par un index unique et un article.

1.3. Création d'un article

La création d'article se fait avec l'appel à la méthode createArticle qui prend le contenu de l'article comme paramètre.

```
34  // Fonction qui permet la création d'article
35  function createArticle(string memory _content) public {
36    uint index = ids.length;
37    ids.push(index);
38    Article memory newArticle = Article(_content);
39    articlesById[index] = newArticle;
40    historyOfArticles[index][0] = newArticle;
41    idsOfHistory[index] = 0;
42  }
```

Figure 3: Création d'un article dans le contrat

- Un nouvel index unique est créé, puis ajouté à au tableau d'index (ligne 36-37).
- Puis l'article est instancié en faisant appel à la struct avec son contenu (ligne 38).
- L'article est ajouté au map en utilisant comme clé l'index instancié juste avant (ligne 39).
- Puis on initialise l'historique du nouvel article, en affectant à l'index 0 l'article qu'on vient de créer (ligne 40).
- Puis on initialise aussi le map contenant les index des dernières modifications on lui affectant un 0 (ligne 41).

1.4. Modification article

```
45  // Fonction qui permet la mise à d'un article
46    function updateArticle(uint _id, string memory _content) public {
47      uint id = idsOfHistory[_id];
48      historyOfArticles[_id][id++] = articlesById[ids[_id]];
49      articlesById[ids[_id]].content = _content;
50      idsOfHistory[_id] = id;
51    }
```

Figure 4: Modification article

La modification d'article se fait par appel à la méthode updateArticle avec l'index et le nouveau contenu de l'article à mettre à jour en paramètres.

- Dans un premier temps en récupère l'index de la dernière modification (ligne 47).
- On ajoute l'article avant de le mettre à jour dans l'historique avec comme clé l'index de la dernière modification +1 (ligne 48).
- On met à jour le contenu de l'article (ligne 49).

- Puis on change la valeur de l'index de la dernière modification ayant été incrémentée de 1 (ligne 50).

1.5. Récupération des article

```
// Fonction qui permet de recuperrer un article avec son id
function articleContent(uint index) public view returns (string memory) {
  return articlesById[index].content;
}

// Fonction qui permet de récuperer la liste des ids
function getAllIds() public view returns (uint[] memory) {
  return ids;
}
```

Figure 5: Récupération des ids et des article

La récupération des articles se fera en deux phases, la première on récupère la liste des indexs de tous les articles avec la méthode getAllIds(), puis on fera appel à la méthode articleContent, pour récupérer le contenu de chaque chaque article.

1.6. Récupération de l'historique d'un article

```
// Fonction qui permet d'envoyer le nombre de fois q'un article a été met à jour
function getHistoricalCount(uint _id) public view returns (uint){
  return idsOfHistory[_id];
}

// Fonction qui permet d'envoyer l'historique d'un article en fonction de l'id de la mise à jour
function getHistorical(uint _idArticle, uint _idUpdate) public view returns (string memory){
  return historyOfArticles[_idArticle][_idUpdate].content;
}
```

Figure 6: Récupération de l'historique d'un article

La récupération de l'historique se fera en deux phases aussi, la première on récupère l'index de la dernière modification d'un article avec la méthode getHistoricalCount, et comme l'index est incrémenté à chaque mise à jour, il représente aussi le nombre de modifications d'un article, puis on fera appel à la méthode getHistorical pour récupérer une des modification antérieur.

2 . Partie front

2.1. Récupération des articles et de leur historique

```
108  const loadArticles = async() =>{
109
110    var recevedArticles = [];
111    var res = [];
112
113    recevedArticles = await contract.methods.getAllIds().call();
114    for (var i = 0; i < recevedArticles.length; i++) {
115      const article = await contract.methods.articleContent(recevedArticles[i]).call();
116      res.push(article);
117
118      var countofHistory = [];
119      var historyRes = [];
120      countofHistory = await contract.methods.getHistoricalCount(i).call();
121
122      for (var j = 0; j < countofHistory; j++) {
123        const history = await contract.methods.getHistorical(i, j).call();
124        historyRes.push(history);
125      }
126      setMyMap(myMap.set(i,historyRes));
127    }
128    setArticles(articles.concat(res));
129  }
```

Figure 7: Récupération des articles et leur historique

- On récupère la liste des ids de tous les articles (ligne 113).
- Pour chaque id, on récupère le contenu de l'article en question, qu'on ajoute dans un état local (ligne 115).
- Puis on récupère le dernier index de modification avec getHistoricalCount().call(), puis on récupérera toutes les modifications antérieur avec la methode getHistorical(indexArticle, indexModification), l'historique sera ajouté à un état local qu'on pourra utiliser pour l'affichage (ligne 120-125).

2.2. Ajout d'article

Les articles sont ajoutés avec un formulaire contenant un textarea et un un input type submit.

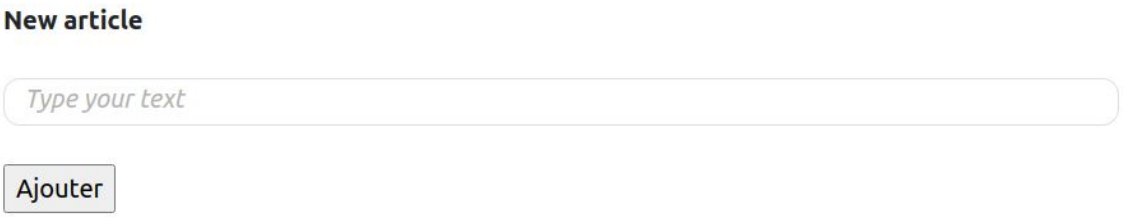

Figure 7: Ajout d'un article

Une fois le bouton ajouter appuyé, la fonction onSubmit sera appelée, cette dernière va récupérer le contenu du textarea, qu'elle le passera en paramètre par la suite au service de création d'article.

```
22  const onSubmit = async (evt) => {
23
24    var content = document.getElementsByClassName(styles.editable)[0].firstElementChild.innerHTML;
25    try {
26      await contract.methods.createArticle(content).send({ from: contract.account });
27    } catch (error) {
28      console.error(error)
29    }
30  }
```

Figure 8: Fonction d'ajout d'article onSubmit

Le service de création d'article est juste un appel à la méthode createArticle exposée par le contrat (ligne 26). Elle envoie une transaction au contrat intelligent et exécute sa méthode, en fournissant l'adresse de source de la transaction.

2.3. Modification article

```
80  const updateArtcile = async(evt) =>{
81    evt.preventDefault();
82    try {
83      await contract.methods.updateArticle(idArticle,updateArticle).send({ from: contract.account });
84      articles[idArticle] = updateArticle;
85      setModal(!modal);
86    } catch (error) {
87      console.error(error)
88    }
89    getHistory(idArticle);
90  }
```

Figure 9: Fonction de modification d'article

- La modification d'article se fait par l'appel à la méthode du contrat updateArticle, avec comme argument l'index de l'article à mettre à jour et le nouveau contenu de ce dernier.

- Puis la l'état local est mis à jour avec le nouveau contenu.
- Puis l'historique est mis à jour en faisant appel à la fonction getHistory.

2.3. Historique d'un article

```
92    const getHistory = async (id) =>{
93      var countofHistory = [];
94      var historyRes = [];
95      countofHistory = await contract.methods.getHistoricalCount(id).call();
96
97      for (var j = 0; j < countofHistory; j++) {
98        const history = await contract.methods.getHistorical(id, j).call();
99        historyRes.push(history);
100     }
101     setMyMap(myMap.set(id,historyRes));
102   }
```

Figure 10: Fonction de récupération de l'historique d'un article

Cette fonction prend l'index de l'article en paramètre puis:

- Appel la méthode exposée par le contrat getHistoricalCount().call(), pour récupérer l'index de la dernière mise à jour.
- Récupérer toutes les modifications antérieures avec la méthode getHistorical(indexArticle, indexModification).cal(), et ajoute de chacune d'elle dans l'état local (ligne 97-101).

3 . Fortmatic

3.1. Intégration de Fortmatic

L'intégration de Fortmatic se fait dans la méthode de connexion, l'intégration va permettre aux utilisateurs existants qui ont installé MetaMask de continuer à utiliser MetaMask sans interruption et de diriger les nouveaux utilisateurs sans MetaMask vers Fortmatic.

```
if (window.ethereum) {
```

Figure 11: Test prèsence MetaMask

La vérification de la présence de MetaMask se fait avec cette ligne de code, si l'utilisateur a installé MetaMask, on exécute ce qui se trouve dans la condition. Si non on exécute ce qui se trouve dans le else.

```
23      const customNodeOptions = {
24        rpcUrl: 'http://127.0.0.1:7545', // your own node url
25        chainId: 1337 // chainId of your own node
26        }
27        const fm = new Fortmatic('pk_test_7DBA38B8B256024A', customNodeOptions);
28        window.web3 = new Web3(fm.getProvider());
29        try {
30          const [account]= await window.web3.eth.getAccounts(); //Fortmatic
31          const contract = new window.web3.eth.Contract(
32            ContractInterface.abi,
33            ContractInterface.networks['5777'].address,
34            { from: account }
35          )
36          dispatch(connectEthereum({ account, contract }))
37        } catch (error) {
38          console.error(error)
39        }
```

Figure 12: Connexion à Fortmatic

- On a ajoute le nœud personnalisé de ganache (ligne 23-26).
- Puis on crée et initialise le provider (ligne 27-28).
- Et on récupère le compte utilisateur (ligne 30).
- Le reste du code ne change pas, par apport à la configuration de MetaMask.

3.2. Test de Fortmatic

Comme on vérifie l'existence de MetaMask en premier, on doit désactiver l'extension MetaMask de chrome, pour pouvoir faire les tests avec Fortmatic. L'intégration de Fortmatic s'est bien faite, au lancement de l'application une authentification est demandée, puis la récupération des articles se passe très bien. Mais lors de l'ajout d'articles ou de modification, Formatic demande de payer une certaine somme pour pouvoir valider la transaction, comme indiqué sur la figure suivante:

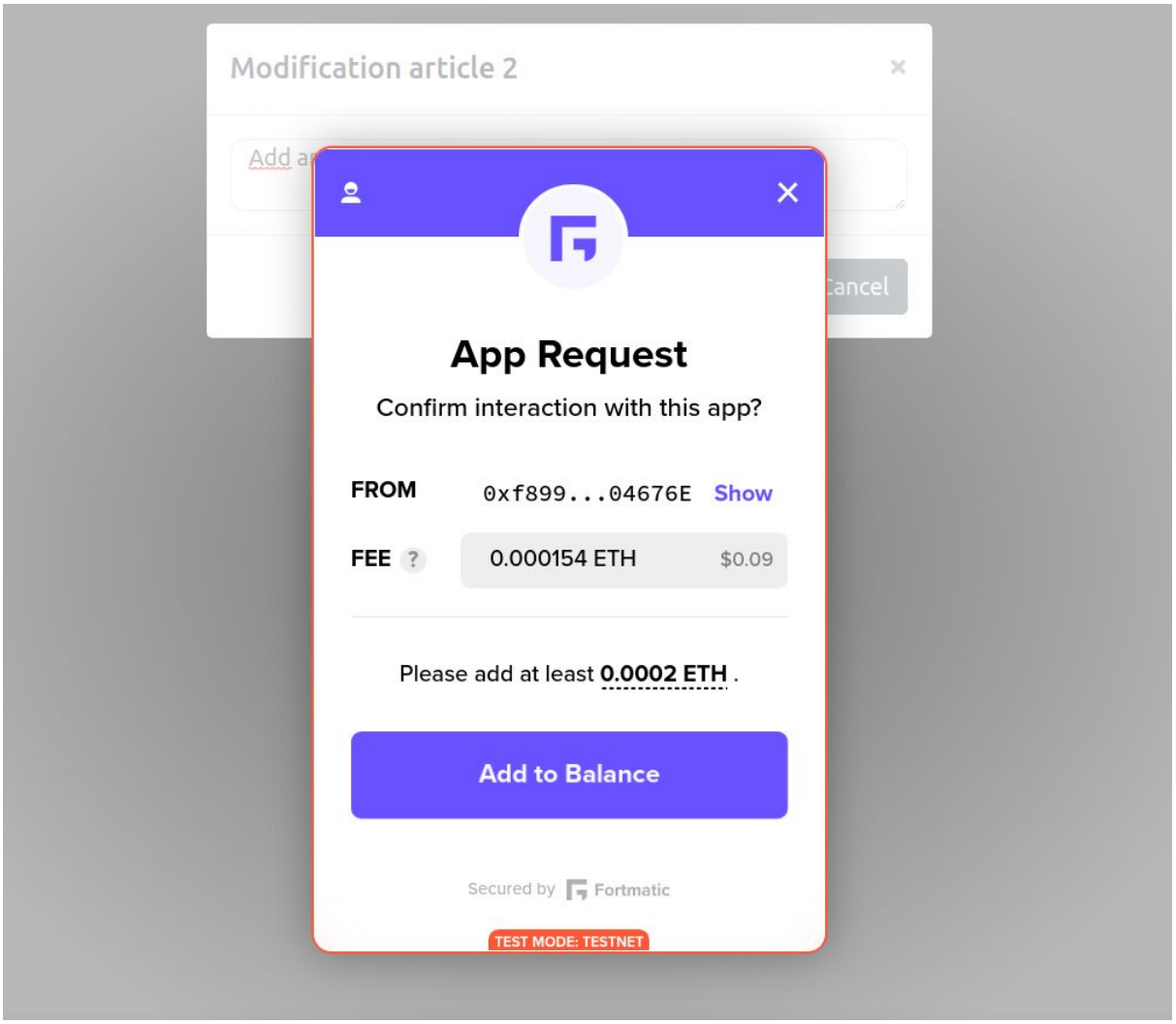

Figure 13: Demande validation transaction Fortmatic

Remarque

- Après chaque modification du contrat, une recompilation est nécessaire.
- La plupart du code a été écrit dans App.js, une séparation des services vers Ethereum.js a été débutée à la fin du jalons, mais par faute de temps cette restructuration n’a pas pu être menée au bout.