

Application décentralisée :

Achat et Vente de Produits basée sur la Blockchain

Encadré par : Mr ALI EL KSIMI

Réalisé par:

NAJLAA JOULALI

SALMA HERMAK

KHADIJA BOUYOUSSEF

AICHA NASIH

Table des matières

<u>INTRODUCTION</u>	3
<u>OBJECTIFS PROJET</u>	3
2.1 OBJECTIF GENERAL	3
2.2 OBJECTIFS SPECIFIQUES	3
<u>ARCHITECTURE DU SYSTEME</u>	4
3.1 COMPOSANTS PRINCIPAUX	4
SMART CONTRACT (BACKEND DECENTRALISE)	4
BLOCKCHAIN LOCALE (GANACHE)	4
INTERFACE UTILISATEUR (FRONTEND)	5
WEB3.JS (PASSERELLE D'INTERACTION BLOCKCHAIN)	5
METAMASK (GESTIONNAIRE DE COMPTES ETHEREUM)	5
DIAGRAMME D'ARCHITECTURE	6
<u>MISE EN PLACE DE L'ENVIRONNEMENT</u>	7
OUTILS NECESSAIRES	7
<u>DEVELOPPEMENT DU SMART CONTRACT</u>	8
5.1. CREATION D'UN PRODUIT	8
OBJECTIF DE LA FONCTIONNALITE	8
STRUCTURE DE DONNEES UTILISEE	8
IMPLEMENTATION SOLIDITY	8
ÉVENEMENT ASSOCIE	9
CONTROLE DE VALIDITE	9
5.2. ACHAT D'UN PRODUIT	9
OBJECTIF DE LA FONCTIONNALITE	9
CONDITIONS DE SÉCURITÉ	10
IMPLEMENTATION SOLIDITY	10
ÉVÉNEMENT ASSOCIÉ	11
SÉCURITÉ DES PAIEMENTS	11
5.3. GESTION ET CONSULTATION DES PRODUITS	12
ACCES AUX PRODUITS PAR ID	12
CONSULTATION DES PRODUITS POSSEDES	12
UTILITE DANS LE CONTEXTE APPLICATIF	12
<u>DEPLOIEMENT SUR GANACHE</u>	13

PREPARATION DU DEPLOIEMENT	13
CREATION DU PROJET TRUFFLE	13
CONFIGURATION DU RESEAU LOCAL DANS TRUFFLE	13
CONFIGURATION DE GANACHE	14
VERIFICATION DE L'ENVIRONNEMENT	15
COMPILATION ET MIGRATION AVEC TRUFFLE	15
COMPILATION DU CONTRAT	15
SCRIPT DE MIGRATION	16
DÉPLOIEMENT SUR GANACHE	16
VERIFICATION DU DEPLOIEMENT	17
VÉRIFICATION DANS GANACHE	17
INTEGRATION AVEC METAMASK	19
CREATION ET IMPORTATION DE COMPTES	19
CONNEXION AU RESEAU LOCAL	21
SIGNATURE DES TRANSACTIONS	22
DEVELOPPEMENT DE L'INTERFACE WEB	23
INTEGRATION DE WEB3.JS	23
INTERACTION AVEC LE CONTRAT	24
 DEPLOIEMENT SUR LE RESEAU SEPOLIA	 25
 CONFIGURATION D'ALCHEMY	 25
CONFIGURATION D'ALCHEMY	25
RECUPERATION DE L'URL RPC :	25
SECURISATION DES VARIABLES SENSIBLES :	25
COMPILATION ET DEPLOIEMENT	26
RESULTAT DU DEPLOIEMENT	26
RESULTAT DE LA COPULATION :	26
RESULTAT DU DEPLOIEMENT	27
APPROVISIONNEMENT DU COMPTE VIA GOOGLE CLOUD FAUCET	27
 FONCTIONNALITES REALISEES	 29
 TEST DES FONCTIONNALITES ET LES RESULTATS	 31

Introduction

Dans le cadre de l'initiation aux technologies de la blockchain, ce projet a pour objectif de concevoir et de mettre en œuvre une **application décentralisée (DApp)** simulant un système de **place de marché numérique**. L'application permet la création, la consultation et l'achat de produits à travers des **smart contracts déployés sur une blockchain Ethereum locale**, simulée par **Ganache**.

Ce projet met en œuvre des outils de développement blockchain comme **Solidity**, **Truffle**, **Ganache**, ainsi que des technologies web modernes telles que **React.js** et **Web3.js** pour la couche front-end et l'interaction avec les contrats intelligents.

Objectifs projet

2.1 Objectif général

Développer une plateforme décentralisée simulant une place de marché numérique, dans laquelle des utilisateurs peuvent interagir (créer, acheter, consulter) autour de produits inscrits dans une blockchain Ethereum, avec une interface web connectée à la blockchain locale via MetaMask et Ganache.

2.2 Objectifs spécifiques

1. **Comprendre le fonctionnement des contrats intelligents Ethereum**
 - Acquérir une maîtrise de base du langage **Solidity**.
 - Apprendre à structurer un contrat pour représenter des entités comme des produits et des transactions commerciales.
 - Intégrer des mécanismes de sécurité (vérification d'adresse, conditions d'achat, accès propriétaire).
2. **Développer un contrat intelligent représentant une marketplace décentralisée**
 - Permettre l'enregistrement des produits (titre, description, prix) par les vendeurs.
 - Gérer la vente : permettre à un acheteur d'acquérir un produit en envoyant des ethers.
 - Mettre à jour dynamiquement l'état du produit après l'achat (statut : disponible ou vendu).
 - Conserver un historique traçable des transactions sur la blockchain.
3. **Déployer et tester les contrats localement dans un environnement sécurisé**
 - Simuler un réseau Ethereum à l'aide de **Ganache**.
 - Utiliser **Truffle** pour compiler, migrer et tester les contrats.
 - Exécuter des scénarios de test réalistes pour valider la robustesse du smart contract.
4. **Créer une interface utilisateur intuitive et fonctionnelle**
 - Utiliser **React.js** pour construire l'interface web moderne.
 - Intégrer **Web3.js** pour interagir avec les smart contracts depuis le navigateur.
 - Assurer la communication en temps réel entre le contrat et l'interface utilisateur.
5. **Configurer un portefeuille et gérer les transactions avec MetaMask**
 - Créer un portefeuille Ethereum avec **MetaMask**.

- Connecter MetaMask au réseau local Ganache.
- Utiliser MetaMask pour signer les transactions d'achat et d'enregistrement de produit.

6. Approfondir la compréhension du cycle de vie complet d'une DApp

- Identifier et comprendre les différentes couches : contrat (backend blockchain), interface (frontend), passerelle (Web3).
- Mettre en œuvre un flux complet, de la création d'un produit jusqu'à la finalisation d'une vente.

Architecture du système

3.1 Composants principaux

Smart Contract (Backend décentralisé)

Le smart contract est le cœur logique de la DApp. Écrit en **Solidity**, il est déployé sur la blockchain Ethereum locale via Ganache.

Rôle fonctionnel :

Gérer la création, la modification, et la suppression des produits listés dans la marketplace.

Assurer le suivi du statut des produits (disponible, vendu, retiré).

Gérer les transactions d'achat : réception des paiements, transfert de propriété, mise à jour de l'état des produits.

Sécurité et immutabilité :

Le code du smart contract est immuable une fois déployé, ce qui garantit que la logique métier ne peut pas être altérée frauduleusement.

Toutes les transactions sont enregistrées de manière transparente et permanente dans la blockchain, assurant traçabilité et confiance.

Gestion des événements :

Le smart contract émet des événements à chaque action importante (exemple : nouveau produit créé, produit vendu). Ces événements sont captés par le frontend pour mise à jour en temps réel de l'interface utilisateur.

Blockchain locale (Ganache)

Ganache est un simulateur blockchain Ethereum local, essentiel pour le développement et les tests rapides sans passer par un réseau public.

Fonctionnalités clés :

- Fournit plusieurs comptes Ethereum préconfigurés avec des ethers fictifs pour simuler des transactions sans coûts réels.
- Affiche en temps réel l'état de la blockchain, les blocs minés, et les transactions.
- Permet de réinitialiser la blockchain locale à tout moment, facilitant le test des modifications.

Avantages :

- Rapidité des transactions, sans latence réseau réelle.

- Environnement sécurisé et isolé du réseau principal, idéal pour le développement et le débogage.

Interface Utilisateur (Frontend)

L'interface utilisateur est développée en **React.js**, un framework JavaScript moderne qui facilite la création de composants interactifs et dynamiques.

- **Fonctionnalités offertes :**
 - Liste dynamique des produits disponibles à la vente avec affichage des détails (titre, description, prix, propriétaire).
 - Formulaire de création d'un nouveau produit avec validation des données avant envoi au smart contract.
 - Interface d'achat permettant à un utilisateur de sélectionner un produit, déclencher la transaction et recevoir un retour sur son succès ou échec.
- **Réactivité :**
 - Utilisation des hooks React pour gérer l'état local, notamment la mise à jour instantanée des produits suite aux événements blockchain.
 - Intégration avec Web3.js pour interagir avec le smart contract et écouter les événements en temps réel.

Web3.js (Passerelle d'interaction blockchain)

Web3.js est une bibliothèque JavaScript qui sert de pont entre le frontend et la blockchain Ethereum.

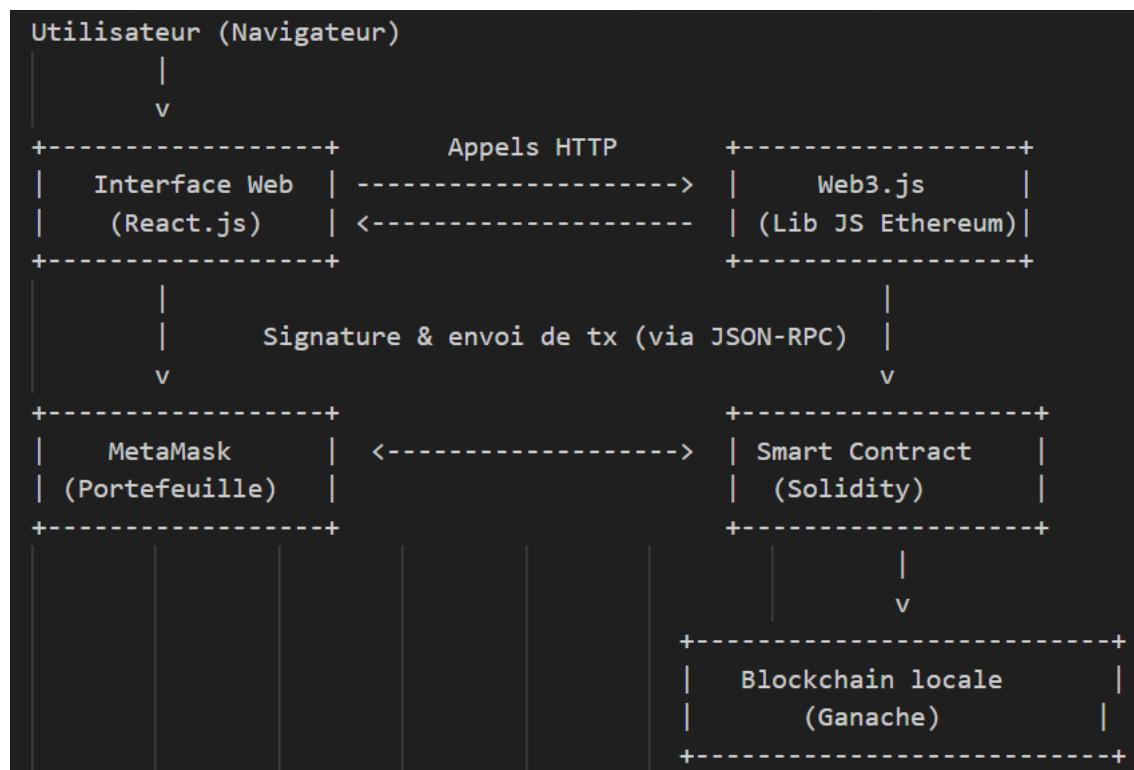
- **Rôle principal :**
 - Permet d'appeler les fonctions du smart contract (lecture et écriture).
 - Gérer les transactions signées par l'utilisateur.
 - Écouter les événements émis par le smart contract pour mettre à jour l'interface utilisateur automatiquement.
- **Fonctionnement :**
 - Web3.js se connecte au provider Ethereum fourni par MetaMask, qui assure la gestion sécurisée des clés privées.
 - Les appels en lecture sont gratuits et immédiats, tandis que les appels en écriture (transactions) nécessitent une signature par l'utilisateur.

MetaMask (Gestionnaire de comptes Ethereum)

MetaMask est une extension de navigateur (Chrome, Firefox, Edge) qui joue le rôle de portefeuille Ethereum et d'interface utilisateur pour la gestion des comptes.

- **Fonctions clés :**
 - Stockage sécurisé des clés privées des utilisateurs, cryptées localement.
 - Gestion multiple de comptes Ethereum (adresses).
 - Signature des transactions initiées par le frontend, garantissant que l'utilisateur approuve chaque opération.
 - Connexion facile à différents réseaux Ethereum, y compris Ganache pour le développement local.
- **Sécurité :**
 - L'utilisateur garde le contrôle exclusif de ses clés privées, aucune information sensible n'est envoyée au serveur distant.
 - Les transactions sont soumises à une confirmation explicite par l'utilisateur avant d'être envoyées à la blockchain.

Diagramme d'architecture



1. Utilisateur (navigateur)

L'utilisateur accède à l'application via son navigateur, où l'interface React.js est chargée. Il peut visualiser les produits, en ajouter de nouveaux ou en acheter.

2. Frontend (React.js)

L'interface en React.js communique avec le smart contract grâce à la bibliothèque **Web3.js**. Elle récupère les données et les affiche, ou bien initie des transactions (création de produit, achat...).

3. Web3.js

Sert de passerelle entre l'application React et la blockchain.

- Elle interroge la blockchain pour afficher les données.
- Elle prépare les transactions à signer.

4. MetaMask

Lorsque l'utilisateur initie une transaction (ex. : acheter un produit), **MetaMask** est déclenché. Il :

- Affiche une fenêtre de confirmation.
- Signe la transaction avec la **clé privée de l'utilisateur**.
- Envoie la transaction à Ganache via Web3.

5. Smart Contract (Solidity)

Une fois la transaction reçue, le **contrat intelligent** l'exécute :

- Vérifie que les conditions sont remplies (par exemple, le bon montant payé).
- Modifie l'état du produit sur la blockchain (par exemple : marqué comme vendu).
- Émet un événement signalant la mise à jour.

6. Ganache (Blockchain locale)

Le réseau Ganache reçoit la transaction, la mine dans un bloc et conserve l'historique.

- Il retourne une confirmation à Web3.js.

- Web3.js capte les **événements du contrat** pour mettre à jour automatiquement le frontend.

Mise en place de l'environnement

Le développement d'une DApp nécessite un environnement complet qui englobe la blockchain, le déploiement des smart contracts, la gestion des comptes, ainsi que l'interface utilisateur. Cette section décrit les **outils nécessaires** et les étapes pour **créer le projet à l'aide de Truffle**.

Outils nécessaires

Outil	Rôle
Node.js	Fournit l'environnement d'exécution JavaScript et le gestionnaire de paquets npm .
Truffle	Framework de développement pour les smart contracts Ethereum. Il permet la compilation, le déploiement et les tests.
Ganache	Blockchain Ethereum locale permettant des tests rapides, avec comptes pré-configurés.
MetaMask	Extension de navigateur qui joue le rôle de portefeuille Ethereum.
Visual Studio Code (ou éditeur similaire)	Pour écrire et organiser le code (Smart Contracts, frontend, scripts, etc.).
React.js	Framework frontend pour construire une interface utilisateur dynamique.
Web3.js	Bibliothèque permettant au frontend d'interagir avec la blockchain.

Développement du smart contract

5.1. Crédit d'un produit

Objectif de la fonctionnalité

La fonction `createProduct` est au cœur de la logique de mise en vente dans la plateforme. Elle permet à tout utilisateur (vendeur) de créer une annonce de produit en renseignant un **titre**, une **description** et un **prix**. Ce produit sera ensuite stocké sur la blockchain sous forme de structure, et deviendra disponible à l'achat pour les autres utilisateurs.

Cette étape matérialise la décentralisation du processus de mise en ligne d'un article, sans avoir recours à un tiers de confiance.

Structure de données utilisée

La structure principale utilisée ici est la suivante :

```
struct Product {  
    uint id;  
    string title;  
    string description;  
    uint price;  
    address payable seller;  
    address owner;  
    bool isSold;  
}
```

Chaque produit est représenté par un identifiant unique (`id`) et contient les informations essentielles pour la vente. Les champs `seller` et `owner` permettent de tracer respectivement l'adresse du créateur et du futur acheteur. Le champ `isSold` sert à marquer l'état de l'article.

Tous les produits sont stockés dans une table de hachage (`mapping`) de type :

```
mapping(uint => Product) public products;
```

Chaque produit est accessible via son identifiant.

Un second `mapping` permet de suivre les produits associés à chaque adresse utilisateur (vendeur ou acheteur) :

```
mapping(address => uint[]) public ownedProducts;
```

Implémentation Solidity

Voici la fonction Solidity concernée, extraite et expliquée :

```
function createProduct(  
    string memory _title,  
    string memory _description,  
    uint _price  
) public {  
    require(bytes(_title).length > 0, "Title cannot be empty");  
    require(bytes(_description).length > 0, "Description cannot be empty");  
    require(_price > 0, "Price must be greater than zero");  
  
    productCount++;
```

```

products[productCount] = Product(
    productCount,
    _title,
    _description,
    _price,
    payable(msg.sender),
    address(0),
    false
);

ownedProducts[msg.sender].push(productCount);

emit ProductCreated(productCount, _title, _description, _price,
msg.sender);
}

```

Explication technique :

- **Validation des entrées** : les champs `_title`, `_description` et `_price` sont vérifiés afin de garantir l'intégrité de l'annonce.
- **Incrémentation** : `productCount` permet d'assurer un identifiant unique.
- **Enregistrement du produit** : la structure est stockée dans `products`.
- **Suivi de l'auteur** : le produit est lié au vendeur via `ownedProducts`.
- **Émission d'un événement** : `ProductCreated` notifie la blockchain d'une nouvelle création.

Événement associé

L'émission d'un événement Solidity permet à l'interface Web ou aux outils d'analyse (comme Remix ou Etherscan) de suivre les actions importantes du contrat :

```

event ProductCreated(
    uint id,
    string title,
    string description,
    uint price,
    address seller
);

```

Celui-ci est émis à chaque fois qu'un produit est ajouté.

Contrôle de validité

L'utilisation des `require` assure :

- La **non-nullité** des champs ;
- L'exclusion des **produits gratuits** ;
- La **sécurité** contre les enregistrements vides ou malicieux.

5.2. Achat d'un produit

Objectif de la fonctionnalité

La fonction `purchaseProduct` permet à un utilisateur d'acheter un produit mis en vente. Cette opération entraîne :

- Le transfert de fonds entre l'acheteur et le vendeur,
- Le changement de propriétaire du produit,
- La mise à jour de l'état du produit (`isSold`).

Cela simule le cœur du modèle de marketplace décentralisée.

Conditions de sécurité

Avant tout traitement, plusieurs vérifications sont effectuées pour garantir la légitimité et la sécurité de la transaction :

```
sécurité de la transaction :  
require(_id > 0 && _id <= productCount, "Product does not exist");  
require(msg.value == product.price, "Incorrect price");  
require(!product.isSold, "Product already sold");  
require(msg.sender != product.seller, "Seller cannot buy own product");
```

Ces require empêchent :

- L'achat d'un produit inexistant,
- Une erreur ou une tentative de fraude sur le montant envoyé,
- Une double vente,
- Un achat par le vendeur lui-même.

Implémentation Solidity

```
function purchaseProduct(uint _id) public payable {  
    Product storage product = products[_id];  
    require(_id > 0 && _id <= productCount, "Product does not exist");  
    require(msg.value == product.price, "Incorrect price");  
    require(!product.isSold, "Product already sold");  
    require(msg.sender != product.seller, "Seller cannot buy own product");  
  
    // Transfert du paiement au vendeur  
    product.seller.transfer(msg.value);  
  
    // Mise à jour de l'état du produit  
    product.owner = msg.sender;  
    product.isSold = true;  
  
    // Enregistrement du produit chez l'acheteur  
    ownedProducts[msg.sender].push(_id);  
  
    emit ProductPurchased(_id, msg.sender, msg.value);  
}
```

Le cœur de la fonction s'organise en deux phases clés :

1. **Transfert des fonds :** Le montant payé par l'acheteur (`msg.value`) est envoyé directement à l'adresse du vendeur (`product.seller`). Cette opération utilise la méthode `transfer`, garantissant une transaction sécurisée et limitant les risques d'attaques comme la ré-entrée.
2. **Transfert de propriété :** Le champ `owner` du produit est mis à jour avec l'adresse de l'acheteur (`msg.sender`). Le booléen `isSold` est positionné à `true` pour indiquer que le produit n'est plus disponible.

Enfin, un événement est émis pour notifier la blockchain et les interfaces externes :

```
emit ProductPurchased(_id, msg.sender, msg.value);
```

Cet événement est essentiel pour la transparence et la synchronisation des données côté front-end.

Étape	Description
1. Chargement du produit	Le produit est récupéré par ID depuis products.
2. Vérification	Les quatre require filtrent les actions invalides.
3. Paiement	L'argent est transféré au vendeur via transfer.
4. Mise à jour	Le nouveau propriétaire est enregistré, et l'état isSold devient true.
5. Historique	Le produit est ajouté à la liste de l'acheteur via ownedProducts.
6. Événement	Un signal ProductPurchased est émis sur la blockchain.

Événement associé

```
event ProductPurchased (
    uint id,
    address buyer,
    uint price
);
```

Cet événement est utile pour :

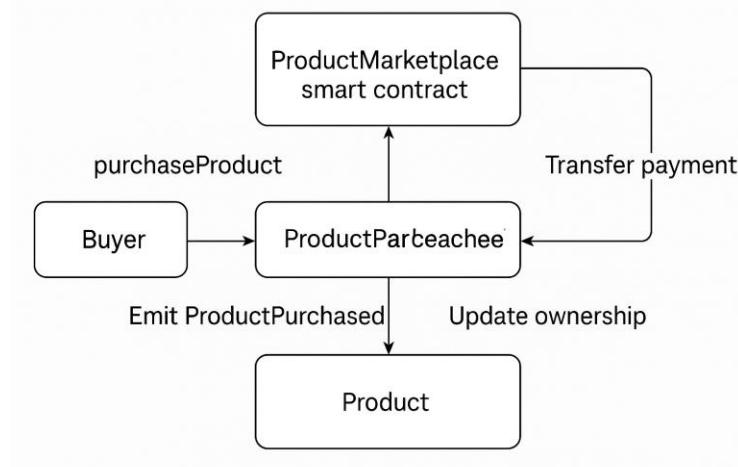
- L'interface Web (actualisation automatique),
- Le suivi de la transaction sur des explorateurs blockchain.

Sécurité des paiements

L'utilisation de `transfer` garantit un transfert sécurisé d'Ether au vendeur. En Solidity $\geq 0.8.0$, cela limite les risques d'attaque de type re-entrancy (notamment grâce à l'absence d'appel récursif ici).

Schéma de flux

Voici un schéma de flux pour illustrer les échanges lors de l'achat :



5.3. Gestion et consultation des produits

Accès aux produits par ID

Le contrat permet d'accéder aux détails d'un produit à partir de son identifiant unique via la fonction `getProductDetails`.

```
function getProductDetails(uint _id) public view returns (
    uint id,
    string memory title,
    string memory description,
    uint price,
    address seller,
    address owner,
    bool isSold
) {
    Product memory product = products[_id];
    return (product.id, product.title, product.description, product.price, product.seller, product.owner, product.isSold);
}
```

- Cette fonction retourne l'ensemble des informations relatives à un produit : son titre, description, prix, vendeur, propriétaire actuel, et son état de vente.
- Elle utilise un mapping `products` qui stocke chaque produit par son ID.
- Le type `view` garantit qu'elle ne modifie pas l'état de la blockchain et n'engendre pas de coûts en gaz lors de son appel.

Consultation des produits possédés

Pour connaître les produits possédés par une adresse Ethereum donnée, le contrat expose la fonction `getOwnedProducts` :

```
function getOwnedProducts(address _owner) public view returns (uint[] memory) {
    return ownedProducts[_owner];
}
```

- Cette fonction renvoie un tableau d'identifiants des produits détenus par l'utilisateur.
- Le mapping `ownedProducts` associe une adresse à une liste d'IDs produits.
- Cela permet à une interface de consulter rapidement l'ensemble des produits qu'un utilisateur vend ou a achetés.

Utilité dans le contexte applicatif

- **Transparence et traçabilité** : Ces fonctions assurent que les utilisateurs peuvent consulter facilement et en toute sécurité les informations produits, garanties par la blockchain.
- **Expérience utilisateur améliorée** : Grâce à l'accès direct aux produits par ID ou par propriétaire, l'interface peut afficher dynamiquement les catalogues personnels et les détails produits.
- **Réduction des coûts d'interaction** : Ces fonctions sont en lecture seule (`view`), ce qui signifie qu'elles ne consomment pas de gaz et peuvent être appelées gratuitement par les utilisateurs.
- **Fiabilité** : Les données sont stockées de manière immuable, évitant toute modification frauduleuse.

Déploiement sur Ganache

Préparation du déploiement

Création du projet Truffle

Avant toute chose, il faut initialiser un nouveau projet Truffle pour gérer le développement et le déploiement du contrat :

```
mkdir Marketplace
cd Marketplace
truffle init
```

Cette commande crée la structure de base du projet, notamment les dossiers contracts, migrations, et test, ainsi que le fichier truffle-config.js essentiel pour la configuration du réseau.

```
MarketPlace/
├── contracts/
│   ├── ProductMarketplace.sol      # contrat intelligent principal
│   └── Migrations.sol            # Contrat de migration
├── migrations/
│   ├── 1_initial_migration.js    # Migration initiale
│   └── 2_deploy_product_marketplace.js # Script de migration
└── test/
    ├── ProductMarketplace.test.js # (Optionnel) tests pour votre contrat
├── .gitignore
└── truffle-config.js           # Fichier de configuration du projet
```

- **contracts/** : dossier contenant tous vos contrats Solidity. Ici, ProductMarketplace.sol est le contrat principal du projet.
- **migrations/** : dossier contenant les scripts de déploiement. 1_initial_migration.js déploie Migrations.sol automatiquement, et vous ajouterez 2_deploy_product_marketplace.js pour déployer votre contrat principal.
- **test/** : dossier destiné aux tests automatiques de vos contrats (en JavaScript ou Solidity).
- **truffle-config.js** : fichier qui configure votre projet Truffle, notamment la connexion à Ganache.

Configuration du réseau local dans Truffle

Avant toute opération de déploiement, il est crucial de configurer correctement l'environnement de développement afin d'assurer une communication fluide entre Truffle, Ganache, et le contrat Solidity. Cette étape comprend la configuration du fichier `truffle-config.js`, la mise en place du réseau local Ganache, et la vérification des comptes disponibles.

Le fichier `truffle-config.js` permet de définir les réseaux sur lesquels les contrats seront déployés. Pour le développement local, nous utilisons Ganache comme blockchain privée.

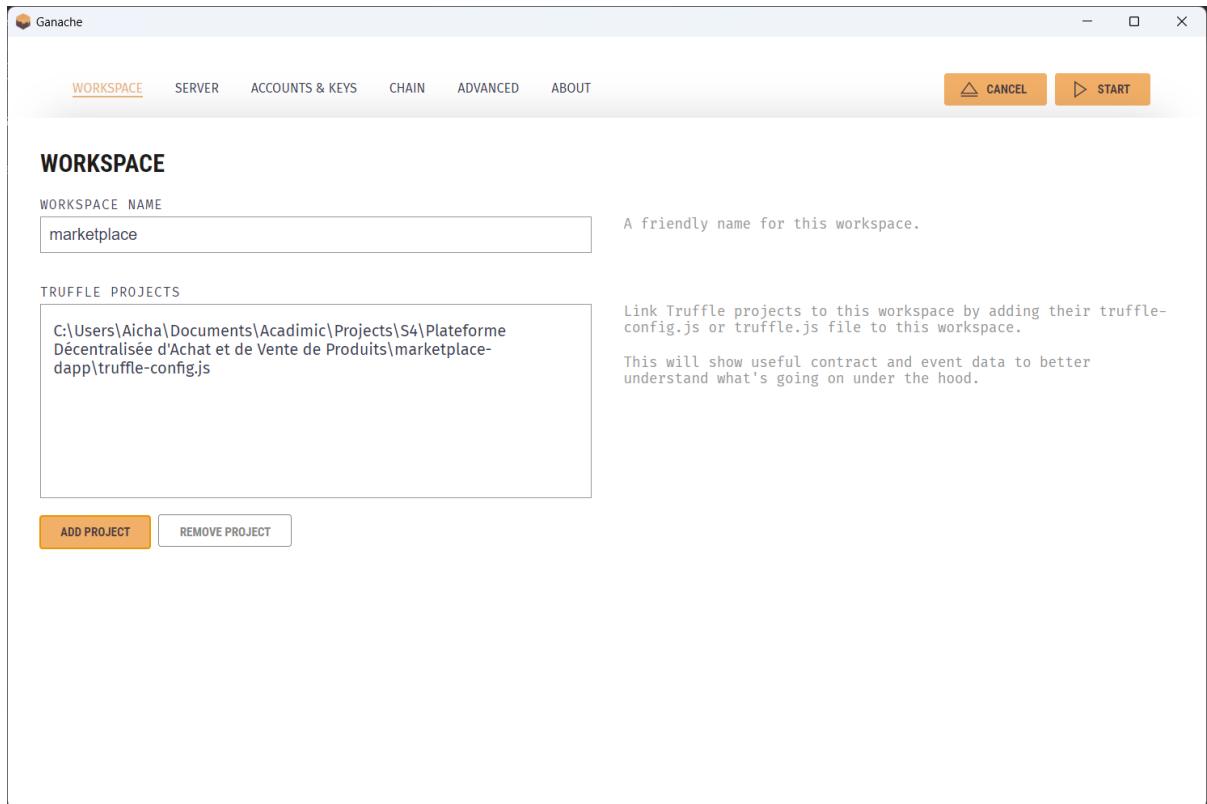
Voici un extrait du fichier configuré :

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",          // Adresse locale de Ganache
      port: 7545,                 // Port de Ganache (à adapter selon votre instance)
      network_id: "*",           // Accepte n'importe quel ID de réseau
```

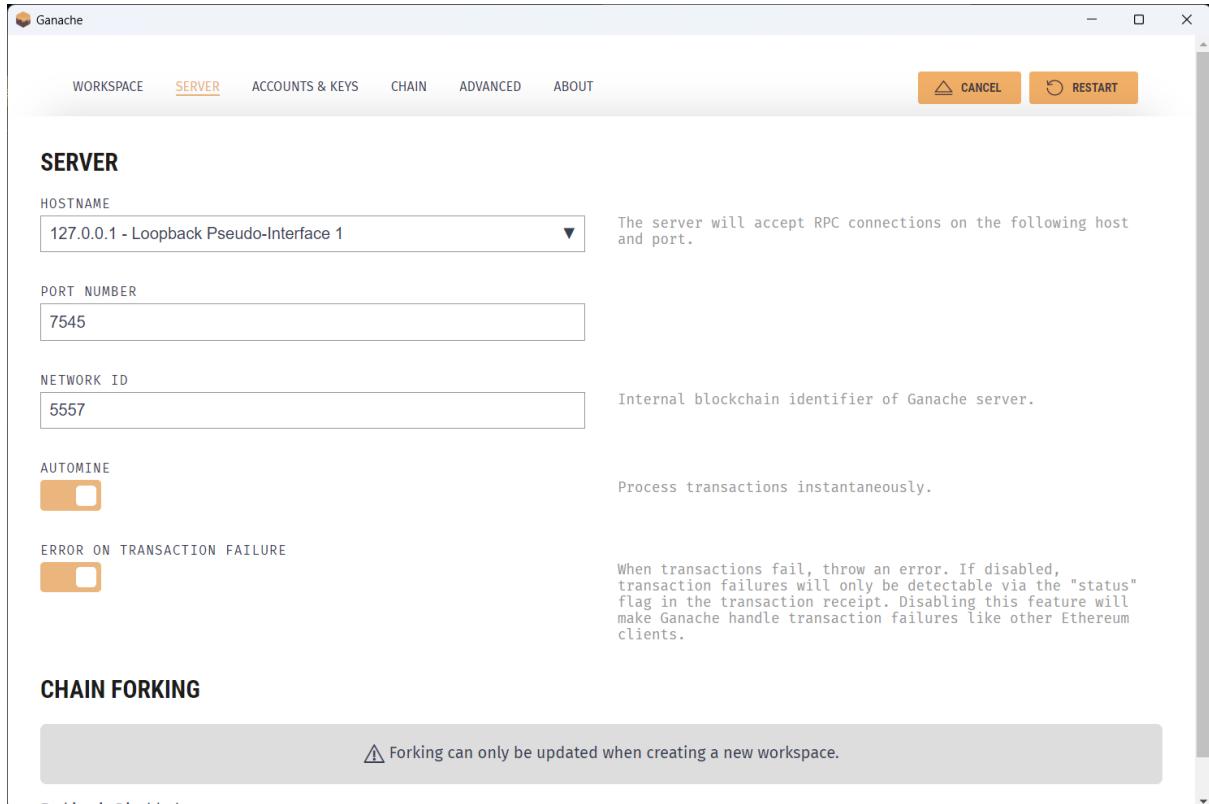
```
        },
        },
        compilers: {
            solc: {
                version: "0.8.0",           // Version du compilateur compatible avec
le contrat
            },
        },
    };
}
```

Configuration de Ganache

- **Lancement de Ganache :** ouvrir l'application Ganache GUI



- **Vérification du port :** s'assurer que Ganache est bien lancé sur le port indiqué dans truffle-config.js (ex. : 7545)



Vérification de l'environnement

Avant de procéder à la compilation :

- S'assurer que Ganache est bien lancé et accessible.
- Vérifier la connectivité de Truffle avec Ganache via `truffle develop`.

```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS node + ⌂ ⌄ ... ^

PS C:\Test\marketplace-dapps> truffle develop
This version of pWS is not compatible with your Node.js build:

Error: Cannot find module '../binaries/uws_win32_x64_127.node'
Require stack:
- C:\Users\Aicha\AppData\Roaming\npm\node_modules\truffle\node_modules\ganache\node_modules\@trufflesuite\uws-js-unofficial\src\uws.js
- C:\Users\Aicha\AppData\Roaming\npm\node_modules\truffle\node_modules\ganache\dist\node\core.js
- C:\Users\Aicha\AppData\Roaming\npm\node_modules\truffle\build\develop.bundled.js
- C:\Users\Aicha\AppData\Roaming\npm\node_modules\truffle\node_modules\original-require\index.js
- C:\Users\Aicha\AppData\Roaming\npm\node_modules\truffle\build\cli.bundled.js
Falling back to a NodeJS implementation; performance may be degraded.

Truffle Develop started at http://127.0.0.1:9545/

Accounts:
(0) 0x11a4383676991b6e64e51f46699983af46267a4
(1) 0x57867f12d475bfc1a61db796587dd6da00bc7
(2) 0xe770fd5d1530c044d7ae278a5ccfd5d930a6e9fe
(3) 0xdcd3d1a83ad4954c8defaefdb8c4e97ff35f9a6e
(4) 0x0c8be5e46aaad7f3e0023dfda6f126981ddee6f0
(5) 0xace7338e82fac33d96a185276be269191496d26
(6) 0xf0dc964cd4ea0ee5e997288254c0814d53f2397e
(7) 0x75d7f5de62ad0edd65353ab926cd972b5a78031d
(8) 0x46c2c1e8f3f5cc78c2d786d172a4bfc065105a8
(9) 0xda92f12a33869014ab9d07746149a5a4629286a4

Private Keys:
(0) daf63b55f8ff4636f354b8b862d2effedbf8321c68ef1989d2e0e0f8af4953
(1) 5a7a50fc9172202f02122e87ddc5d1533577ea5b94596ded055e1caf2e8a559
```

Compilation et migration avec Truffle

Une fois l'environnement de développement correctement configuré et Ganache lancé, nous pouvons compiler et déployer le contrat sur la blockchain locale via Truffle.

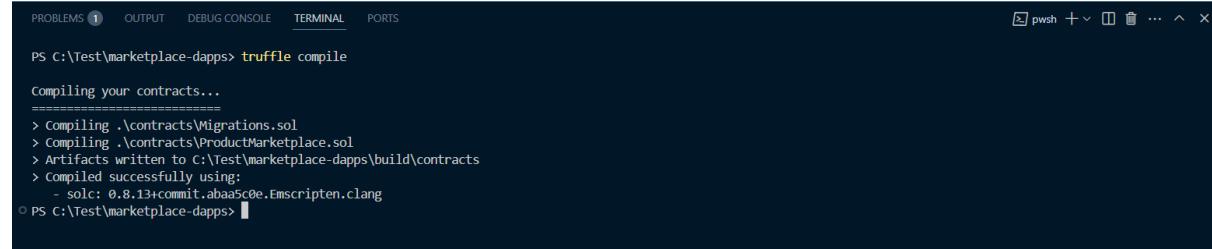
Compilation du contrat

La commande suivante permet de compiler les contrats Solidity définis dans le dossier contracts/ :

```
truffle compile
```

Cette étape :

- Vérifie la syntaxe du code.
- Produit des fichiers **bytecode** et **ABI** dans le répertoire build/contracts.



```
PS C:\Test\marketplace-dapps> truffle compile
Compiling your contracts...
=====
> Compiling ./contracts\Migrations.sol
> Compiling ./contracts\ProductMarketplace.sol
> Artifacts written to C:\Test\marketplace-dapps\build\contracts
> Compiled successfully using:
- solc: 0.8.13+commit.abaa5c0e.Emscripten.clang
○ PS C:\Test\marketplace-dapps>
```

Script de migration

Truffle utilise des scripts de migration pour automatiser le déploiement. Le fichier migrations/2_deploy_contracts.js doit contenir :

```
const ProductMarketplace = artifacts.require("ProductMarketplace");

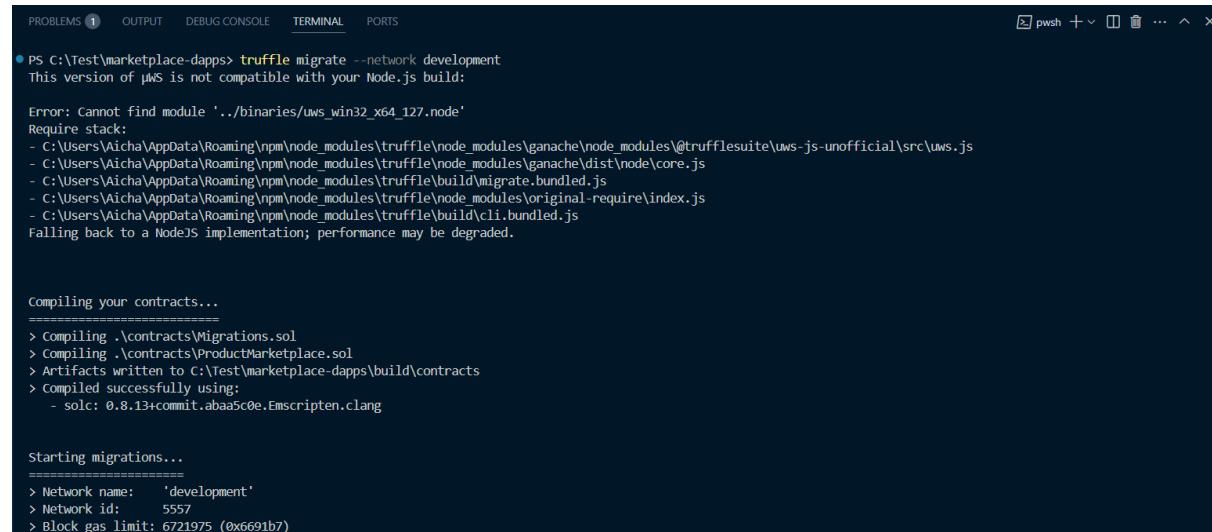
module.exports = function (deployer) {
  deployer.deploy(ProductMarketplace);
};
```

Ce script :

- Charge l'artefact du contrat (artifacts.require).
- Déploie automatiquement le contrat lors de la commande truffle migrate.

Déploiement sur Ganache

Le déploiement se fait via :



```
PS C:\Test\marketplace-dapps> truffle migrate --network development
This version of pWS is not compatible with your Node.js build:

Error: Cannot find module '../binaries/uws_win32_x64_127.node'
Require stack:
- C:\Users\Aicha\AppData\Roaming\npm\node_modules\truffle\node_modules\ganache\node_modules@\trufflesuite\uws-js-unofficial\src\uws.js
- C:\Users\Aicha\AppData\Roaming\npm\node_modules\truffle\node_modules\ganache\dist\node\core.js
- C:\Users\Aicha\AppData\Roaming\npm\node_modules\truffle\build\migrate.bundled.js
- C:\Users\Aicha\AppData\Roaming\npm\node_modules\truffle\node_modules\original-require\index.js
- C:\Users\Aicha\AppData\Roaming\npm\node_modules\truffle\build\cli.bundled.js
Falling back to a NodeJS implementation; performance may be degraded.

Compiling your contracts...
=====
> Compiling ./contracts\Migrations.sol
> Compiling ./contracts\ProductMarketplace.sol
> Artifacts written to C:\Test\marketplace-dapps\build\contracts
> Compiled successfully using:
- solc: 0.8.13+commit.abaa5c0e.Emscripten.clang

Starting migrations...
=====
> Network name:    'development'
> Network id:      5557
> Block gas limit: 6721975 (0x6691b7)
```

Résultat attendu :

- Adresse du contrat déployé.
- Adresses des comptes utilisés (souvent le premier pour le déploiement).
- Confirmation que le contrat a été inséré dans un bloc.

```

2_deploy_product_marketplace.js
=====
Deploying 'ProductMarketplace'
-----
> transaction hash: 0x31bb44f1c142c7f03bc0bdeb6f8100177dfcb7d5eea5189b70b68f37edca6c5a
> Blocks: 0
    Seconds: 0
> contract address: 0x93c64752b2846e88cc637839a1c33e02680849b6
> block number: 73
> block timestamp: 1747269703
> account: 0x257fb38fa3fd37d26e2d771b0dc648830adcecf
> balance: 99.955750305707948642
> gas used: 1372965 (0x14f325)
> gas price: 2.500129275 gwei
> value sent: 0 ETH
> total cost: 0.003432589990050375 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost: 0.003432589990050375 ETH

```

Vérification du déploiement

Après le déploiement du contrat via Truffle, il est essentiel de valider que tout s'est déroulé correctement et que le contrat est bien accessible sur la blockchain locale.

Vérification dans Ganache

- **Adresse du contrat :**

Dans Ganache, sous l'onglet **Contracts** ou **Transactions**, on peut retrouver l'adresse du contrat déployé. Cette adresse est utilisée par l'application front-end pour interagir avec le contrat.

- **Transactions de déploiement :**

La transaction liée au déploiement est visible dans l'historique. Elle indique le hash de la transaction, le bloc dans lequel elle a été minée et le gas consommé.

- **Solde des comptes :**

Le compte utilisé pour le déploiement montre une diminution de son solde, correspondant au coût du gas dépensé pour enregistrer le contrat sur la blockchain.

The screenshot shows the Ganache interface with the following details:

- Accounts:** CURRENT BLOCK 74, GAS PRICE 2000000000, GAS LIMIT 6721975, HARDFORK MERGE, NETWORK ID 5557, RPC SERVER HTTP://127.0.0.1:7545, MINING STATUS AUTOMINING.
- Logs:** WORKSPACE MARKETPLACE
- Search Bar:** SEARCH FOR BLOCK NUMBERS OR TX HASHES
- Switch and Settings:** SWITCH, GEAR icon

Marketplace-dapps C:\Test\marketplace-dapps

NAME	ADDRESS	TX COUNT	DEPLOYED
Migrations	0xd4439aE7Ba5DDdB7499752d77477d7934E6D2ac8	1	DEPLOYED
ProductMarketplace	0x93C64752b2846E88CC637839A1C33e0268DB49b6	0	DEPLOYED

Intégration avec MetaMask

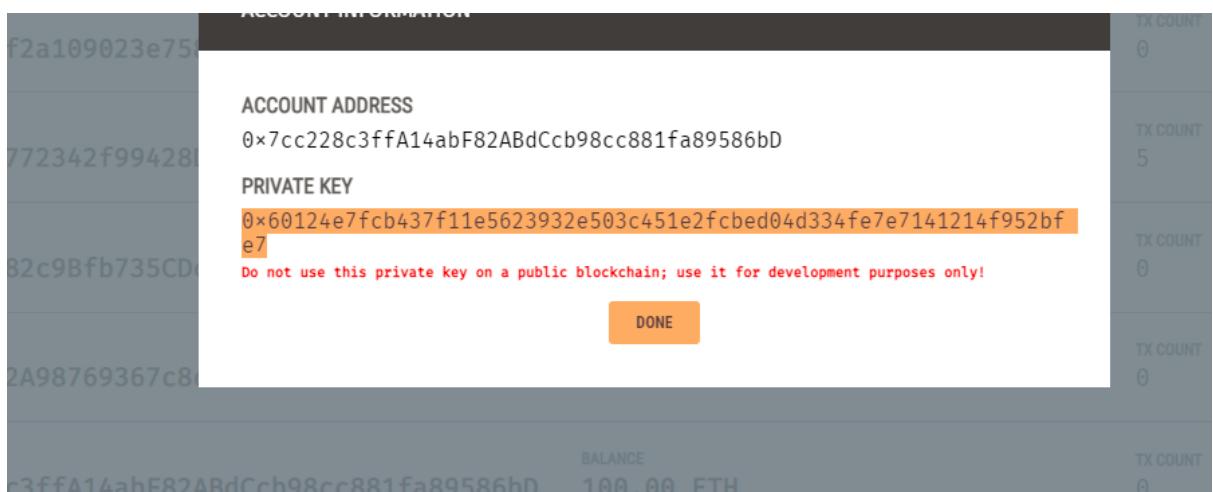
L'intégration de MetaMask à l'environnement de développement est essentielle pour simuler les interactions d'un utilisateur avec le smart contract via une interface web. MetaMask permet de gérer les comptes Ethereum, de signer des transactions, et d'interagir avec des contrats déployés localement (notamment via Ganache).

Création et importation de comptes

MetaMask offre la possibilité de créer de nouveaux comptes ou d'importer ceux générés automatiquement par Ganache.

- ◆ Création de compte dans MetaMask

Dans MetaMask, cliquer sur "Créer un compte".



Dans Ganache, on accéde à l'onglet [Accounts](#). On Copie la clé privée d'un des comptes générés.



Add account



Create a new account

Ethereum account

Import a wallet or account

Secret Recovery Phrase

Private Key

Connect an account

Hardware wallet

Dans MetaMask, on va dans "Importer un compte" puis on colle la clé privée.

Imported accounts won't be associated with your
MetaMask Secret Recovery Phrase. Learn more
about imported accounts [here](#)

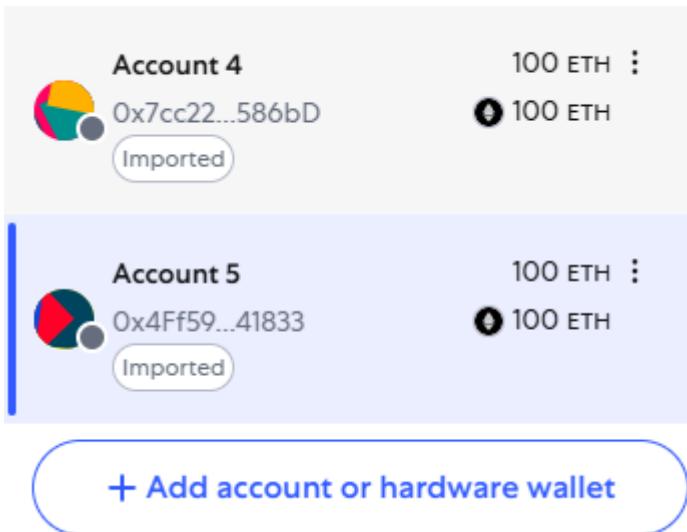
Select Type

Private Key ▾

Enter your private key string here:

[Cancel](#)

[Import](#)



Une fois importé, le compte apparaît avec un solde de 100 ETH simulés.

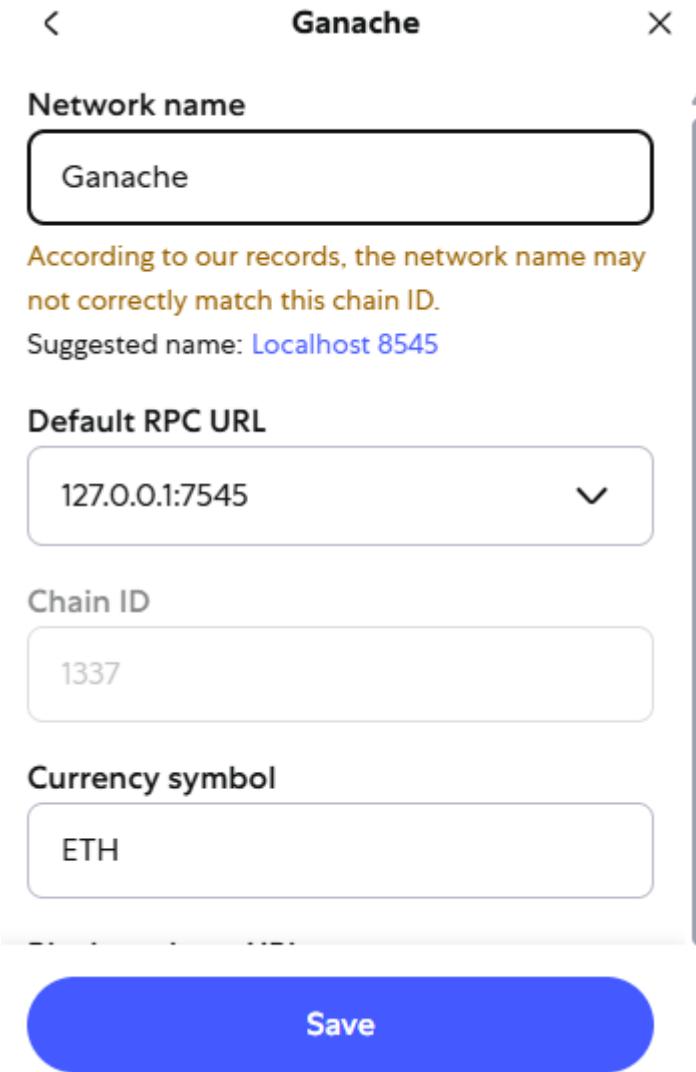
Connexion au réseau local

Pour que MetaMask puisse interagir avec le contrat déployé via Ganache, il est nécessaire de connecter MetaMask au réseau local Ganache.

Configuration manuelle du réseau

- Dans MetaMask : Paramètres > Réseaux > Ajouter un réseau.
- Remplir les champs suivants avec les informations fournies par Ganache :

Champ	Valeur
Nom du réseau	Ganache Local
Nouvelle URL RPC	http://127.0.0.1:7545
ID de chaîne	1337 (ou 5777 selon la version Ganache)
Symbole	ETH



Configuration du réseau local Ganache dans MetaMask

Une fois connecté, MetaMask pourra signer et envoyer des transactions sur ce réseau de développement local.

Signature des transactions

MetaMask joue un rôle central dans l'interaction avec les smart contracts : chaque action utilisateur (achat, création de produit, etc.) déclenche une transaction que MetaMask doit signer.

Fonctionnement :

- Lorsqu'un utilisateur interagit avec l'application (ex : achat via un bouton), une transaction est générée via Web3.js ou Ethers.js.
- MetaMask s'ouvre automatiquement pour **confirmer** la transaction.
- Après validation :

- La transaction est envoyée à Ganache.
- Le contrat effectue les opérations demandées.
- Les soldes et états sont mis à jour.

Développement de l'interface Web

L'interface utilisateur permet aux utilisateurs d'interagir facilement avec la blockchain sans avoir besoin de comprendre les détails techniques. Pour ce faire, nous utilisons **React.js** pour créer une interface dynamique, et **Web3.js** pour relier le frontend à notre smart contract déployé sur Ganache.

Intégration de Web3.js

Création du projet React

```
npx create-react-app frontend
```

```
cd frontend
```

Installer Web3.js

```
npm install web3
```

Rôle : Web3.js est une bibliothèque JavaScript qui permet à ton application React de communiquer avec la blockchain Ethereum (via Ganache). Elle est essentielle pour interagir avec les contrats intelligents, envoyer des transactions, lire l'état du contrat, etc.

Créer un fichier de connexion à Web3

```
javascript
```

```
import Web3 from "web3";
```

```
...
```

Rôle :

Vérifie si l'utilisateur a MetaMask installé.

Crée une instance de Web3 connectée à MetaMask (et donc à Ganache).

Demande à l'utilisateur l'autorisation d'accéder à son compte (signature de transactions).

Résultat :

Cette configuration permet à React d'envoyer des requêtes à la blockchain via MetaMask.

Importer Web3 dans App

```
javascript
```

```
import web3 from "./web3";
```

```
.js
```

Rôle : Permet d'utiliser l'objet web3 (configuré dans l'étape précédente) dans toute l'application React. Grâce à cet objet, tu peux interagir avec Ethereum.

Vérifier la connexion

javascript

```
const accounts = await web3.eth.getAccounts();
```

Rôle :

Récupère les comptes Ethereum disponibles (fournis par MetaMask/Ganache).

Affiche l'adresse du compte actuellement connecté.

C'est la première vérification que la communication avec MetaMask/Ganache fonctionne bien.

Interaction avec le contrat

Permettre à votre interface React de **lire** ou **écrire** des données sur la blockchain via un contrat déployé.

Déploiement sur le réseau Sepolia

Configuration d'Alchemy

Configuration d'Alchemy

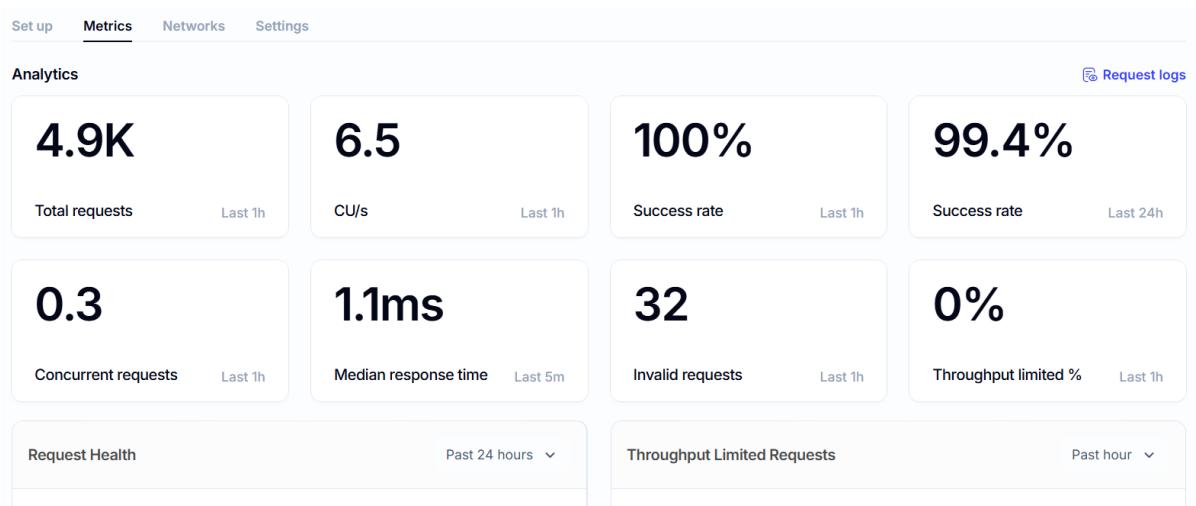
Afin de déployer notre DApp sur le réseau de test **Sepolia**, nous avons utilisé **Alchemy**, une plate-forme de développement Web3 offrant un accès fiable et optimisé à Ethereum via une API RPC.

Voici les étapes suivies :

1. Création du projet :

Sur le site <https://www.alchemy.com/>, nous avons créé un projet avec les paramètres suivants :

- **Nom du projet** : marketplace-dapps
- **Réseau** : Ethereum → Sepolia Testnet



Récupération de l'URL RPC :

Une fois le projet créé, une URL du type suivant nous a été fournie :

`https://eth-sepolia.g.alchemy.com/v2/1ywJ3dQuo1SZDQ1TH_rNAZs2ii1H2MX6`

Sécurisation des variables sensibles :

L'URL ainsi que notre clé privée (compte MetaMask utilisé pour les déploiements) ont été stockées dans un fichier .env, comme recommandé :

Configuration du réseau avec Alchemy

Pour déployer notre smart contract sur le réseau de test Sepolia, nous avons utilisé Alchemy, une plate-forme Web3 qui fournit des nœuds Ethereum accessibles à distance. La configuration a été réalisée via le fichier truffle-config.js, en utilisant la bibliothèque @truffle/hdwallet-provider pour signer les transactions avec notre clé privée.

L'URL fournie par Alchemy a été stockée dans un fichier .env pour des raisons de sécurité, avec également notre clé privée. Voici un extrait de la configuration :

```
require('dotenv').config();
const HDWalletProvider = require('@truffle/hdwallet-provider');

module.exports = {
```

```

networks: {
  sepolia: {
    provider: () =>
      new HDWalletProvider({
        privateKeys: [process.env.PRIVATE_KEY],
        providerOrUrl: process.env.alchemy_URL,
      }),
    network_id: 11155111, // ID du réseau Sepolia
    confirmations: 2,
    timeoutBlocks: 500,
    skipDryRun: true,
  },
  // ...
},
compilers: {
  solc: {
    version: "0.8.13",
  }
},
};

```

Et le fichier .env associé :

```

alchemy_URL=https://eth-sepolia.g.alchemy.com/v2/1ywJ3dQuo1SZDQ1TH_rNAZs2ii1H2MX6
PRIVATE_KEY=95bd373a256df4bab6d77b23875e9e5fb3edfdd4dcda7f673067e4a6b43bf73

```

Compilation et déploiement

La compilation s'est effectuée avec succès :

```
truffle compile
```

Le déploiement sur Sepolia a été lancé avec :

```
truffle migrate --network sepolia
```

Résultat du déploiement

Résultat de la copulation :

Le smart contract a été déployé avec succès et une transaction confirmée sur la blockchain. Le hash de la transaction et l'adresse du contrat sont visibles via l'explorateur Sepolia (Etherscan) :

```

Compiling your contracts...
=====
> Compiling .\contracts\Migrations.sol
> Compiling .\contracts\ProductMarketplace.sol
> Artifacts written to C:\Test\marketplace-dapps\build\contracts
> Compiled successfully using:
  - solc: 0.8.13+commit.abaa5c0e.Emscripten clang

```

Résultat du déploiement

```
2_deploy_product_marketplace.js
=====
Deploying 'ProductMarketplace'
-----
> transaction hash: 0xc299e58421099db4254bd03d4270e304568a3c4d1826a7a83d1d0cb991df5f26
> Blocks: 1 Seconds: 9
> contract address: 0xaCD9e1fE81276e498d1bd15BA94B4ab7c20FCb01
> block number: 8333763
> block timestamp: 1747334208
> account: 0xA1D7c88C45cFB2f32f032ee0B80a56e51fc31753
> balance: 0.037784538165570569
> gas used: 1373351 (0x14f4a7)
> gas price: 4.568468778 gwei
> value sent: 0 ETH
> total cost: 0.00627411164735078 ETH

Pausing for 2 confirmations...

-----
> confirmation number: 1 (block: 8333764)
> confirmation number: 2 (block: 8333765)
> Saving migration to chain.
> Saving artifacts
-----
> Total cost: 0.00627411164735078 ETH
```

Avec Alchemy comme nœud Ethereum et le faucet pour les fonds de test, nous avons pu :

- Déployer notre smart contract avec truffle migrate --network sepolia
- Interagir avec la blockchain en temps réel via MetaMask
- Visualiser les transactions sur Sepolia Etherscan

Approvisionnement du compte via Google Cloud Faucet

Pour exécuter des transactions sur Sepolia (même en test), notre compte devait posséder de l'ETH de test. Nous avons utilisé le **Google Cloud Sepolia Faucet**, accessible ici :

<https://faucet.quicknode.com/ethereum/sepolia>

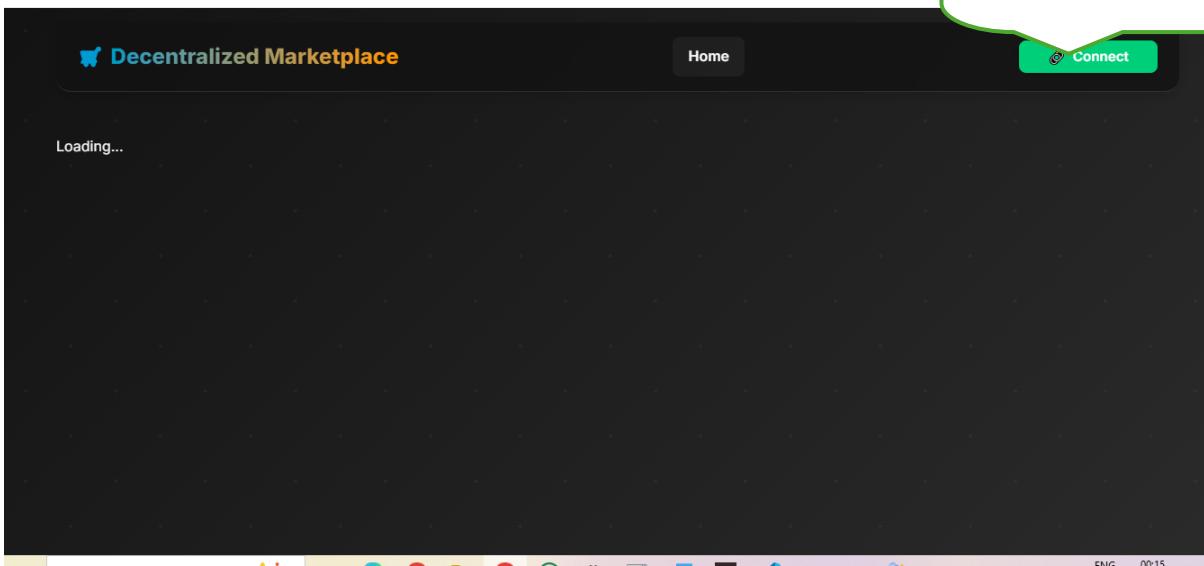
Étapes :

1. **Connexion avec MetaMask** : Nous avons connecté notre portefeuille au site faucet.
2. **Choix du réseau** : Sepolia
3. **Demande d'ETH** : Après vérification Google (reCAPTCHA), le faucet nous a envoyé **0.05 SepoliaETH de test** à l'adresse du compte.

The screenshot shows the "Ethereum Sepolia Faucet" interface. At the top, it says "Get free Sepolia ETH sent directly to your wallet. Brought to you by [Google Cloud for Web3](#)". Below that is a dropdown menu labeled "Select network*" with "Ethereum Sepolia" selected. A note below the dropdown says "*required". There is also a note "Wallet address or ENS name*" followed by a text input field containing the address "0x633c9919030BOF5214b0c522e55f3849d7C508cD". Below the input field is a note "Enter the account address or ENS name where you want to receive tokens". At the bottom is a blue button labeled "Receive 0.05 Sepolia ETH".

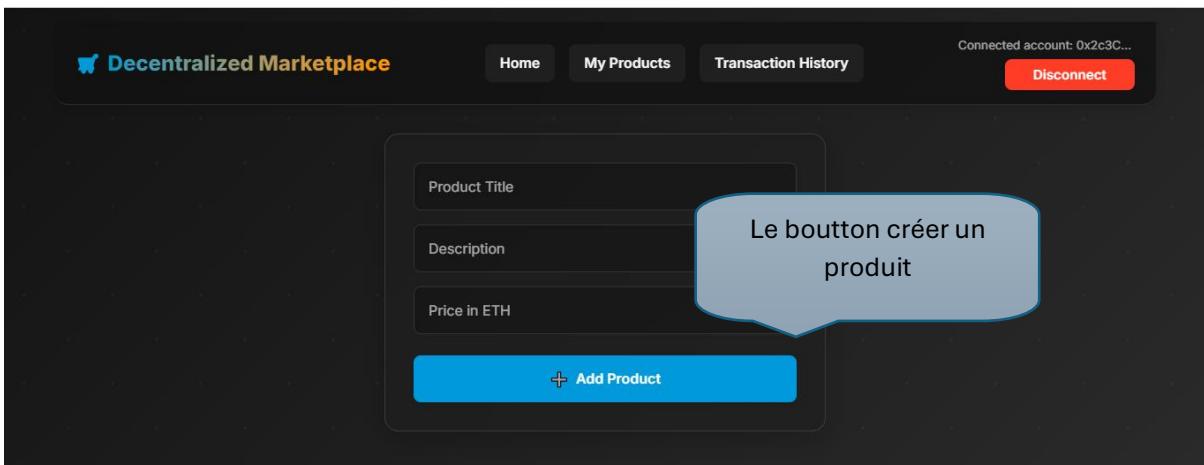
Fonctionnalités réalisées

Voici d'abord Home avant connexion

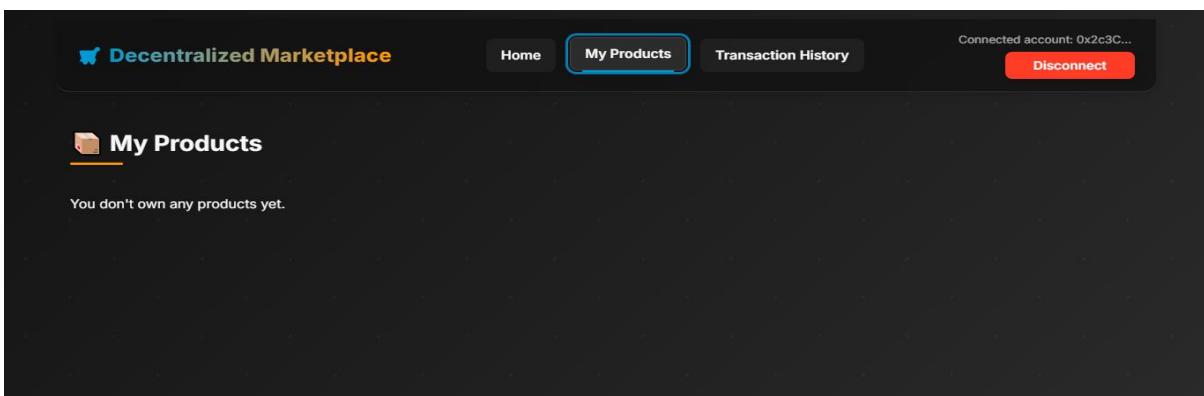


Après la connexion :

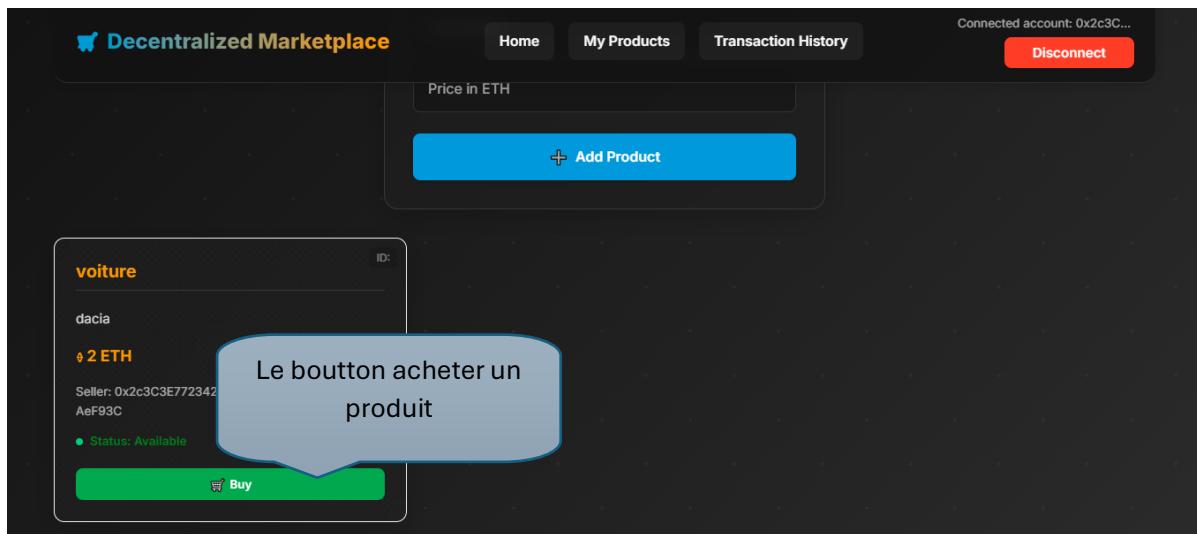
1. Créer un produit



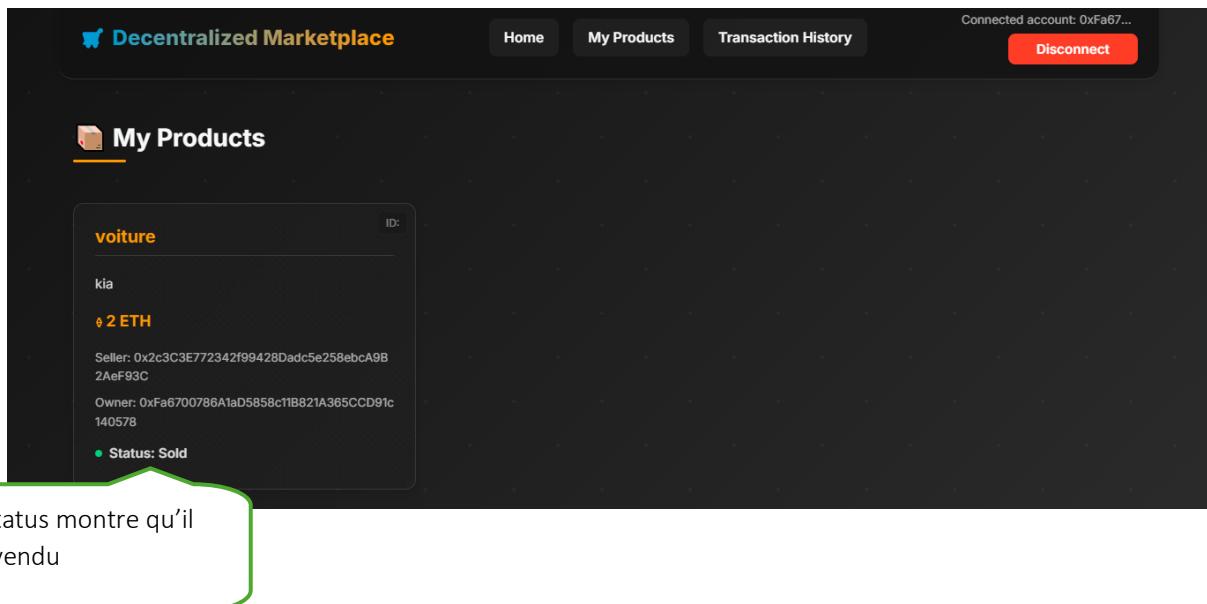
2. Afficher les produits disponibles



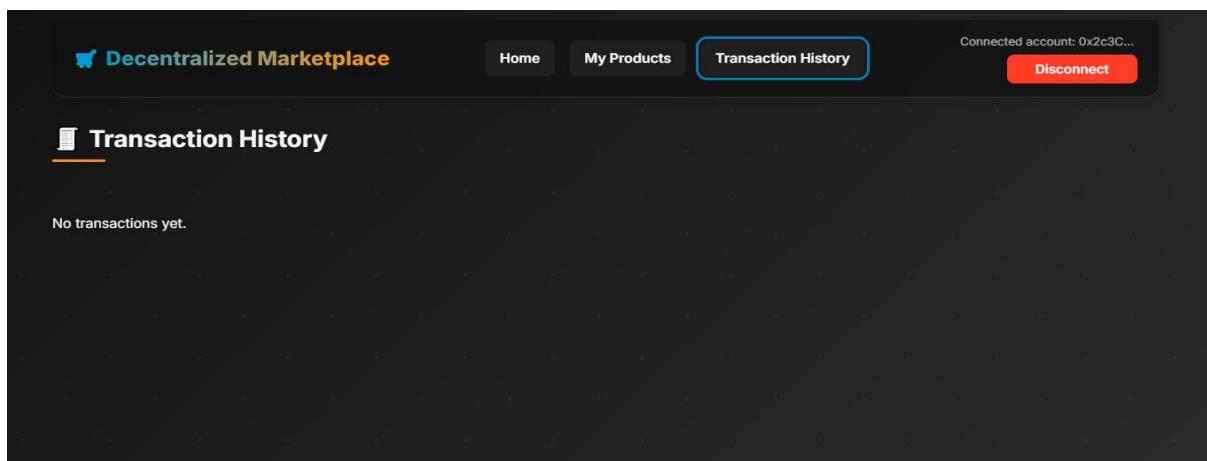
3. Acheter un produit



4. Marquer un produit comme vendu



5. Historique des transactions

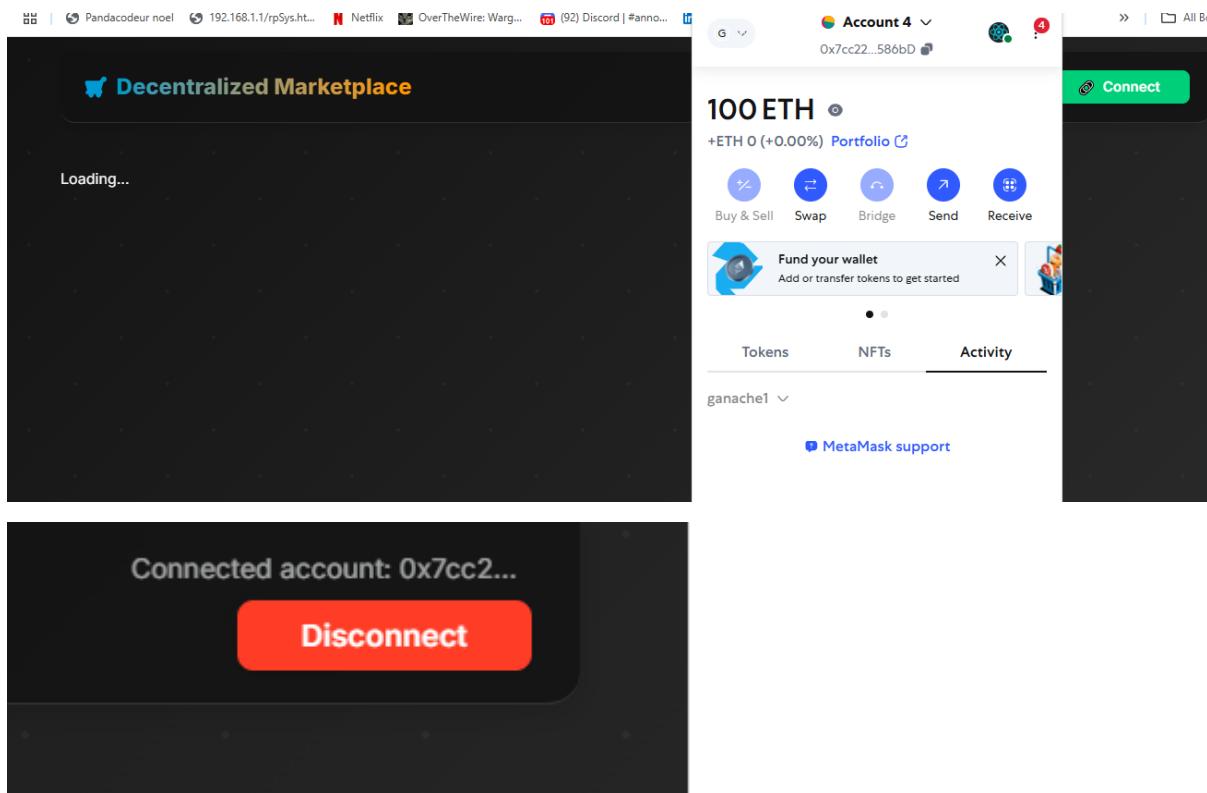


Test des fonctionnalités et les résultats

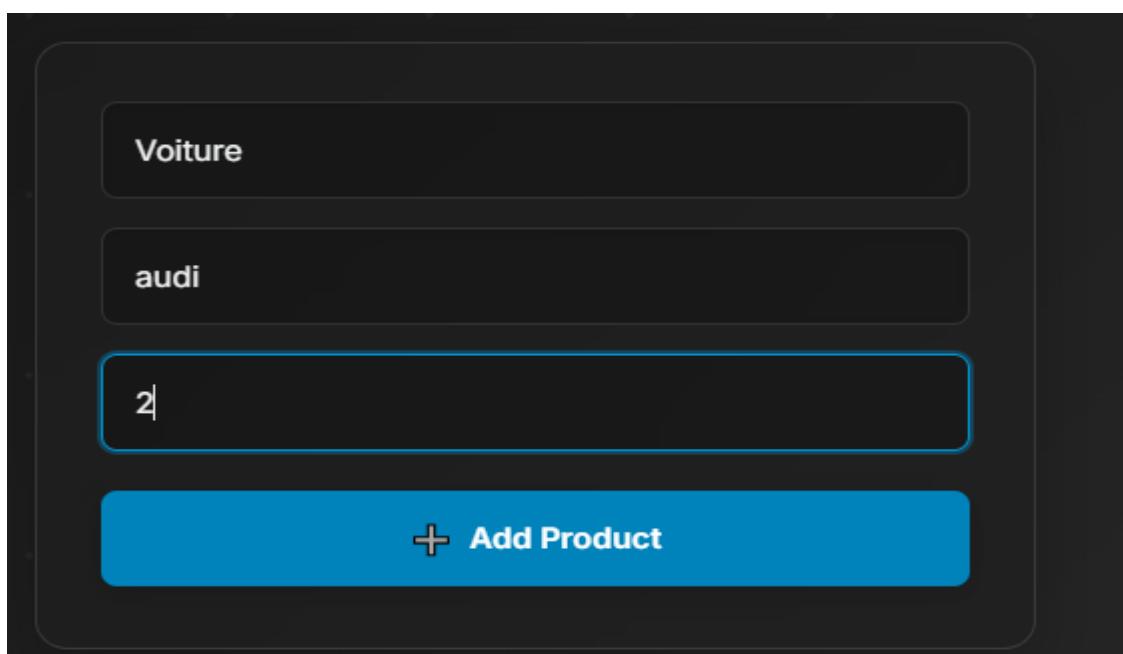
I. Test local sur Ganache :

1- Créer un produit

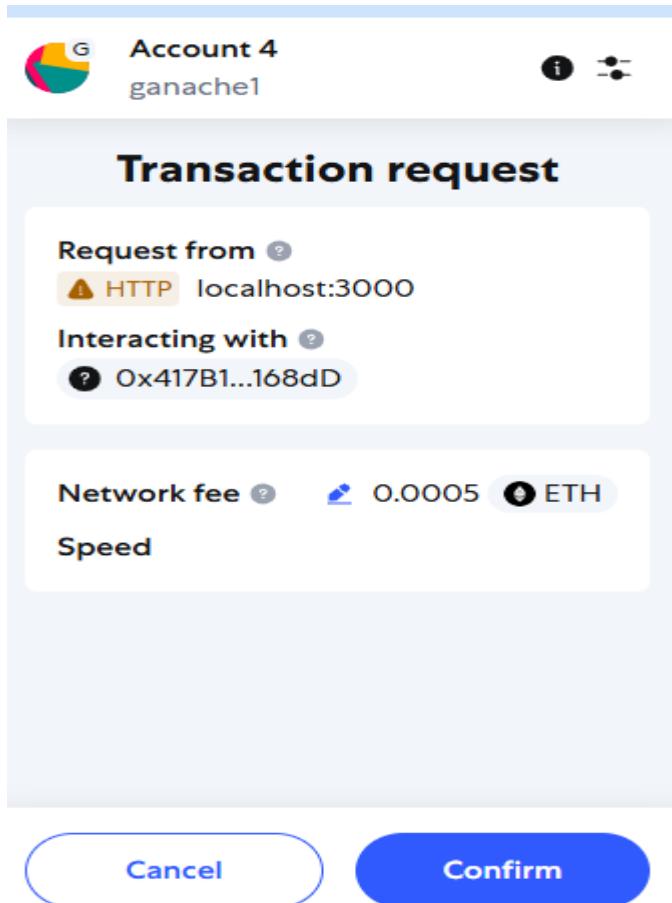
On se connecte avec compte 4



Maintenant on créer un produit :



Après on confirme la transaction avec metamask :



Le produit est ajouté la liste :

A screenshot of the "Decentralized Marketplace" web application. The header includes the logo, "Decentralized Marketplace", navigation links for "Home", "My Products", "Transaction History", and a "Disconnect" button. The "Connected account: 0x7cc2..." is also displayed. The main content area is titled "My Products" and shows a single product entry for a "Voiture":

Voiture
ID: audi
0.2 ETH
Seller: 0x7cc228c3ffA14abFB2ABdCcb98cc881fa89586bd
Owner: 0x00
Status: Available

2- Afficher les produits disponibles

La liste des produits disponibles est mise à jour .

The screenshot shows the 'Decentralized Marketplace' interface. At the top, there are tabs for 'Home', 'My Products', and 'Transaction History'. A connected account '0x7cc2...' is shown with a 'Disconnect' button. Below this, a search bar says 'Price in ETH' and a blue button says '+ Add Product'. Three product cards are displayed:

- voiture** (dacia) - Price: 2 ETH, Status: Available, Buy button.
- voiture** (kia) - Price: 2 ETH, Seller: 0x2c3C3E772342f99428Dadc5e258ebcA9B2AeF93C, Status: Sold, Unavailable button.
- Voiture** (audi) - Price: 2 ETH, Seller: 0x7cc228c3ffA14abF82ABdCcb98cc881fa89586bd, Status: Available, Buy button.

3- Acheter un produit

Maintenant on se connecte avec compte 5 pour acheter ce produit du compte 4.

The screenshot shows the 'Decentralized Marketplace' interface with a connected account '0x4Ff59...'. Below it, a MetaMask wallet interface is visible, showing 100 ETH in the portfolio. The wallet interface includes buttons for 'Buy & Sell', 'Swap', 'Bridge', 'Send', and 'Receive'. A 'Fund your wallet' button is also present. The 'Activity' tab is selected, showing recent activity from 'ganache1'. A 'MetaMask support' link is at the bottom.

On achète (Voiture, audi, 2eth) et comfirme la transaction avec metamask :

The screenshot shows a web browser window for a "Decentralized Marketplace". On the left, there's a list of products: "voiture" (dacia, 2 ETH, Status: Available), "voiture" (kia, 2 ETH, Status: Sold), and "Voiture" (audi, 2 ETH, Status: Available). On the right, a "Transaction request" sidebar is open, showing details for a transaction from "Account 5 (ganache1)" to "OX417B1...168dD". The transaction amount is 2 ETH, and the network fee is 0.0007 ETH. The "Speed" button is highlighted. At the bottom of the sidebar are "Cancel" and "Confirm" buttons.

Le produit est ajoutee a la liste :

The screenshot shows the "My Products" section of the marketplace. It displays the same "Voiture" product as before, but with a different status: "Status: Sold". The seller and owner information remains the same. The "Connected account" is listed as 0x4Ff59... at the top right.

4- Marquer un produit comme vendu

On revient au compte 4 et le produit est marqué vendu.

The screenshot shows the "My Products" section again, with the same "Voiture" product now listed as "Status: Sold" once more. The seller and owner information is identical. The "Connected account" is still 0x4Ff59A2A98769367c8ca5565c8E207805.

5- Historique des transactions

Voici la transaction de compte 4 avec compte 5 du produit (Voiture ,audi,2eth).

The screenshot shows a dark-themed web application interface. At the top, there's a navigation bar with tabs for "Home", "My Products", and "Transaction History". A red "Disconnect" button is on the far right. The main content area is titled "Transaction History" and has a sub-section titled "Voiture". It displays a single transaction entry: "audi" with "2 ETH". Below this, it shows the seller's address (0x7cc2...), the buyer's address (0x4F59A2A98769367c8ca5565c8E2078056d41833), and a green status indicator "Status: Sold".

II. Test sur Etherscan :

Les 4 premières fonctions sont testées de la même façon que ganache

Pour l'historique des transactions :

Compte de Seller sur Etherscan :

The screenshot shows the Etherscan interface for the address 0xA1D7c88C45CfB2f32f032ee0B80a56e51fc31753. The top navigation bar includes "Home", "Blockchain", "Tokens", "NFTs", and "More". The main content area is divided into three sections: "Overview", "More Info", and "Multichain Info". The "Overview" section shows an ETH balance of 0.033300439112405108 ETH. The "More Info" section shows the latest transaction sent was 1 min ago, funded by 0xf192e0D7...DE0c6F734 at tx 0xe3fe2eb7673... The "Multichain Info" section says N/A. Below these, a table lists the "Latest 11 from a total of 11 transactions". The table columns are "Transaction Hash", "Method", "Block", "Age", "From", "To", "Amount", and "Txn Fee". The transactions listed are all related to product creation and completion, with amounts mostly at 0 ETH.

Transaction Hash	Method	Block	Age	From	To	Amount	Txn Fee
0x57cd88fae4d...	Create Product	8334096	1 min ago	0xA1D7c88C...51fc31753	0xaCD9e1fE...7c20FCb01	0 ETH	0.00111655
0xe0bff402687...	Create Product	8334052	10 mins ago	0xA1D7c88C...51fc31753	0xaCD9e1fE...7c20FCb01	0 ETH	0.00113671
0x2a014a7d5d...	Create Product	8334050	10 mins ago	0xA1D7c88C...51fc31753	0xaCD9e1fE...7c20FCb01	0 ETH	0.00112273
0x90f01e10af1...	Create Product	8333870	46 mins ago	0xA1D7c88C...51fc31753	0xaCD9e1fE...7c20FCb01	0 ETH	0.00107573
0xcdaf37d4d69...	Set Completed	8333766	1 hr ago	0xA1D7c88C...51fc31753	0x40cca2E9...F8F31965e	0 ETH	0.00013235
0xc299e58421...	0x60806040	8333763	1 hr ago	0xA1D7c88C...51fc31753	Contract Creation	0 ETH	0.00627411

Compte de Buyer sur Etherscan :

The screenshot shows the Etherscan interface for the address 0x9215950761960c696f9229d6f1BBC6853AcC41AE. The 'Transactions' tab is selected, displaying three recent transactions:

Transaction Hash	Method	Block	From	To	Value	Gas	
0x12ebce1fb96...	Create Product	8334213	3 hrs ago	0x92159507...53AcC41AE	0xCD9e1fE...7c20FCb01	0 ETH	0.00090009
0x9ada8bb55...	Purchase Pro...	8334098	3 hrs ago	0x92159507...53AcC41AE	0xCD9e1fE...7c20FCb01	0.0001 ETH	0.00067965
0x44c5997bef5...	Transfer	8334085	3 hrs ago	0x52f1984C...2E7aCEedD	0x92159507...53AcC41AE	0.05 ETH	0.00007666

II. Publication du contrat sur Etherscan :

Le contrat est publié sur etherscan :

The screenshot shows the Etherscan interface for the contract 0x8a61F7cF2961AE242989a101f827446aC280048B. The 'Contract' tab is selected, showing the following events:

Event	Description	Block	From	To	Value	Gas
1. createProduct (0xa1422209)	Connected - Web3 [0x048e...d0d5]	8334085	0x92159507...53AcC41AE	0xCD9e1fE...7c20FCb01	0 ETH	0.00090009
2. purchaseProduct (0x38e76a03)		8334085	0x52f1984C...2E7aCEedD	0x92159507...53AcC41AE	0.05 ETH	0.00007666

Test d'ajout de produit :

The screenshot shows the Etherscan interface for the same contract 0x8a61F7cF2961AE242989a101f827446aC280048B. The 'Contract' tab is selected, showing the following events:

Event	Description	Block	From	To	Value	Gas
1. createProduct (0xa1422209)	Connected - Web3 [0x048e...d0d5]	8334085	0x92159507...53AcC41AE	0xCD9e1fE...7c20FCb01	0 ETH	0.00090009
2. purchaseProduct (0x38e76a03)		8334085	0x52f1984C...2E7aCEedD	0x92159507...53AcC41AE	0.05 ETH	0.00007666

On a choisi un produit (voiture , audi , 100000) et on a confirmé la transaction :

The screenshot shows a web browser window with multiple tabs open. In the foreground, a Metamask transaction request overlay is displayed. The transaction details are as follows:

- Request from: sepolia.etherscan.io
- Interacting with: 0x8a61F...0048B
- Method: Create Product
- Network fee: 0.0018 SepoliaETH (< ETH 0.01)
- Speed: Market ~12 sec

The transaction itself is for creating a product with the following parameters:

1. createProduct (0xa1422209)
 - _title (string): voiture
 - _description (string): audi
 - _price (uint256): 1000000000000
2. purchaseProduct (0x38e76a03)

At the bottom right of the transaction request overlay are "Cancel" and "Confirm" buttons.

La transaction est confirmée :

The screenshot shows a web browser window with the Etherscan interface. The transaction has been confirmed, and a confirmation message is displayed in a modal window:

Confirmed transaction
Transaction 7 confirmed! View on Sepolia Etherscan

The Etherscan interface shows the transaction details again, including the product creation and purchase steps. At the bottom left, there are "Write" and "View your transaction" buttons.

Vérification de la transaction :

[This is a Sepolia Testnet transaction only]	
② Transaction Hash:	0x4e4d8be679b263fd09e25df73ccceb110877afecfcc27c6ded41c80fc20377c5 ⓘ
② Status:	Success
② Block:	8340139 1 Block Confirmation
② Timestamp:	21 secs ago (May-16-2025 03:59:48 PM UTC)
② From:	0x048e9576D7D33A495351d94a74FD1FdaB8e3D0d5 ⓘ
② To:	0x8a61F7cF2961AE242989a101f827446aC280048B ⓘ ✓
② Value:	0 ETH
② Transaction Fee:	0.001759951932565248 ETH
② Gas Price:	10.011273984 Gwei (0.000000010011273984 ETH)

Conclusion :

Dans le cadre de ce projet, nous avons conçu et développé une application décentralisée (DApp) permettant l'achat et la vente de produits via des transactions sécurisées sur la blockchain. Cette plateforme, réalisée à l'aide de Visual Studio Code, Truffle, React, Ganache, Metamask, Ethers.js, le réseau de test Sepolia et Etherscan, permet aux vendeurs d'enregistrer des produits avec un titre, une description et un prix, tandis que les acheteurs peuvent parcourir les produits disponibles et effectuer des achats en Ether. Chaque transaction est enregistrée de manière transparente et traçable sur la blockchain, et les produits achetés sont automatiquement marqués comme vendus. Ce projet met en évidence les avantages de la technologie blockchain appliquée au e-commerce, notamment la sécurité, la transparence et l'absence d'intermédiaires, tout en nous offrant l'opportunité de renforcer nos compétences en développement Web3 et en intégration de solutions décentralisées.