**Rapport de Projet : DApp de Gestion de Propriétés Immobilières avec React, Hardhat, Ethereum et IPFS**

**Table des Matières :**

1. Introduction et Objectifs du Projet
2. Architecture Globale de la DApp
3. Technologies et Outils Utilisés
   * Backend (Smart Contract & Déploiement)
   * Stockage de Données Décentralisé (IPFS)
   * Frontend (Interface Utilisateur)
4. Développement du Smart Contract (PropertyRegistry.sol)
   * Structure et Logique Métier
   * Fonctionnalités Clés
5. Déploiement du Smart Contract
   * Utilisation de Hardhat
   * Script de Déploiement
6. Gestion des Documents avec IPFS
   * Configuration et Utilisation
7. Développement de l'Interface Utilisateur (Frontend)
   * Choix de Vite et React
   * Intégration de shadcn/ui et Tailwind CSS
   * Structure des Composants
   * Interaction avec MetaMask et le Smart Contract
   * Interaction avec IPFS
8. Fonctionnalités Implémentées
9. Conclusion et Apprentissages
10. Annexe : Commandes Clés Utilisées

**1. Introduction et Objectifs du Projet**

Ce projet consistait à développer une application décentralisée (DApp) pour la gestion de titres de propriété immobilière. L'objectif principal était de créer un système transparent, sécurisé et traçable pour l'enregistrement et le transfert de propriétés, en s'appuyant sur la blockchain Ethereum pour la logique de propriété et IPFS pour le stockage des documents associés.

Les objectifs spécifiques incluaient :

* Permettre l'enregistrement de nouvelles propriétés par un administrateur.
* Assurer la consultation publique des informations des propriétés.
* Permettre le transfert sécurisé de propriété entre utilisateurs.
* Stocker les documents légaux (actes, plans, images) de manière décentralisée et infalsifiable sur IPFS.
* Fournir une interface utilisateur moderne et intuitive pour interagir avec ces fonctionnalités via MetaMask.

**2. Architecture Globale de la DApp**

L'application est structurée en trois couches principales :

* **Smart Contract (Blockchain Ethereum) :** Sert de "notaire" décentralisé et de registre de vérité pour les titres de propriété. Il gère la logique d'ajout, de transfert et de consultation des propriétés.
* **IPFS (InterPlanetary File System) :** Utilisé pour le stockage décentralisé des documents volumineux associés à chaque propriété. Le smart contract ne stocke que les Content Identifiers (CIDs) pointant vers ces documents.
* **Frontend (React avec Vite) :** L'interface web à travers laquelle les utilisateurs interagissent avec le smart contract (via MetaMask) et visualisent les données des propriétés et leurs documents (via la gateway IPFS).

**3. Technologies et Outils Utilisés**

Une sélection de technologies modernes et éprouvées a été faite pour chaque couche de l'application :

* **3.1 Backend (Smart Contract & Déploiement) :**
  + **Solidity (^0.8.9) :** Langage de programmation orienté objet de haut niveau pour l'écriture des smart contracts sur Ethereum. Il offre un contrôle précis sur la logique et l'état du contrat.
  + **Hardhat :** Environnement de développement Ethereum complet.
    - *Pourquoi ?* Facilite la compilation, le déploiement, les tests, et l'exécution d'un nœud Ethereum local pour un développement rapide et itératif. Intègre ethers.js.
  + **Ethers.js (v6.x) :** Bibliothèque JavaScript pour interagir avec la blockchain Ethereum, utilisée dans les scripts Hardhat et le frontend.
  + **OpenZeppelin Contracts (v5.x) :** Bibliothèque de contrats modulaires, réutilisables et sécurisés (notamment Ownable.sol) qui nous a fourni un systeme de gestion de proprietaire et les modifiers onlyowner.
* **3.2 Stockage de Données Décentralisé (IPFS) :**
  + **Kubo via Docker :** Implémentation de référence d'IPFS, conteneurisée avec Docker pour une gestion facile du nœud local.
    - *Pourquoi IPFS ?* Solution de stockage décentralisée, adressage par contenu (garantit l'intégrité des données), idéale pour les fichiers volumineux que l'on ne souhaite pas stocker directement sur la blockchain (coût prohibitif en gaz).
  + **ipfs-http-client (JavaScript) :** Bibliothèque pour interagir avec l'API HTTP d'un nœud IPFS depuis le frontend.
    - *Pourquoi ?* Permet à l'application React d'uploader des fichiers sur le nœud IPFS local de l'utilisateur (ou un nœud distant configuré).
* **3.3 Frontend (Interface Utilisateur) :**
  + **Vite (latest) :** Outil de build frontend moderne et rapide.
    - *Pourquoi ?* Expérience de développement nettement plus rapide que Create React App grâce à son serveur de développement basé sur les modules ES natifs et le HMR performant.
  + **React (v19.x) :** Bibliothèque JavaScript populaire pour la construction d'interfaces utilisateur déclaratives et basées sur les composants.
    - *Pourquoi ?* Vaste écosystème, gestion d'état efficace (Hooks), et approche composant modulaire.
  + **Tailwind CSS (v4.x alpha) :** Framework CSS "utility-first" pour un prototypage et un design rapides et personnalisables.
    - *Pourquoi ?* Permet de styliser directement dans le HTML/JSX sans écrire de CSS personnalisé extensif, favorise la cohérence et la maintenabilité du style. La v4 alpha a été explorée pour ses promesses de performance.
  + **shadcn/ui (latest) :** Collection de composants React réutilisables, accessibles et esthétiques, construits sur Radix UI et Tailwind CSS. Les composants sont copiés dans le projet, offrant une personnalisation maximale.
  + **sonner (via shadcn/ui) :** Bibliothèque de notifications (toasts) React légère et élégante.
    - *Pourquoi ?* Remplacement moderne du composant toast déprécié de shadcn/ui, offrant une meilleure expérience pour les feedbacks utilisateur.
  + **lucide-react :** Bibliothèque d'icônes SVG.
    - *Pourquoi ?* Icônes claires, personnalisables et légères, bien intégrées avec l'écosystème React/Tailwind.
  + **MetaMask :** Extension de navigateur servant de portefeuille Ethereum et d'injecteur de provider Web3.gère les clés privées et la signature des transactions.

**4. Développement du Smart Contract (PropertyRegistry.sol)**

Le smart contract est le cœur immuable de l'application.

* **Structure Property :** Définit les attributs d'un bien immobilier (ID unique, propriétaire actuel, hash IPFS des documents, description, historique des propriétaires).
* **Gestion des IDs :** Un compteur interne (\_propertyIdCounter) assure l'unicité des IDs de propriété.
* **Stockage :** Un mapping (properties) lie chaque ID à sa structure Property correspondante.
* **Contrôle d'Accès :** Le contrat hérite de Ownable.sol d'OpenZeppelin. La fonction addProperty est restreinte au propriétaire du contrat (l'administrateur). La fonction transferProperty ne peut être appelée que par le propriétaire actuel du bien concerné (msg.sender == currentOwner).
* **Fonctionnalités Clés :**
  + addProperty(address \_initialOwner, string memory \_ipfsHash, string memory \_description) : Ajoute un bien.
  + transferProperty(uint \_propertyId, address \_newOwner) : Transfère un bien et met à jour l'historique.
  + getProperty(uint \_propertyId) : Récupère les détails d'un bien.
  + getPropertyHistory(uint \_propertyId) : Récupère l'historique des propriétaires.
  + getTotalProperties() : Retourne le nombre total de biens.
* **Événements :** PropertyAdded et PropertyTransferred sont émis pour permettre le suivi des activités importantes sur la blockchain.

**5. Déploiement du Smart Contract**

* **Hardhat :** Utilisé pour compiler le contrat Solidity (npx hardhat compile), générant l'ABI et le bytecode.
* **Nœud Local Hardhat :** Un réseau Ethereum local a été lancé avec npx hardhat node pour un développement et des tests rapides sans coût de gaz réel.
* **Script de Déploiement (scripts/deploy.js) :** Un script JavaScript utilisant ethers.js (fourni par Hardhat) a été écrit pour :
  1. Obtenir un signataire (le compte deployer).
  2. Obtenir la ContractFactory pour PropertyRegistry.
  3. Déployer le contrat, en passant l'adresse du deployer au constructeur Ownable (nécessaire pour OpenZeppelin v5+).
  4. Attendre la confirmation du déploiement (waitForDeployment()).
  5. Afficher l'adresse du contrat déployé, cruciale pour la configuration du frontend.  
     Le script a été exécuté avec npx hardhat run scripts/deploy.js --network localhost.

**6. Gestion des Documents avec IPFS**

* Un nœud IPFS (Kubo) local a été exécuté via Docker, configuré avec un docker-compose.yml pour exposer l'API (port 5001) et la gateway (port 8080).
* Les documents d'une propriété (multiples fichiers : images, PDF) sont regroupés et uploadés sur IPFS en tant que **dossier unique**.
* L'opération d'upload (réalisée dans le frontend avec ipfs.addAll(files, { wrapWithDirectory: true }) via ipfs-http-client) retourne un CID unique pour ce dossier.
* Ce CID de dossier est ensuite stocké dans le champ ipfsHash du smart contract.
* Les documents sont accessibles via la gateway IPFS locale (ex: http://localhost:8080/ipfs/CID\_DU\_DOSSIER/nom\_du\_fichier.pdf).

**7. Développement de l'Interface Utilisateur (Frontend)**

* **Initialisation avec Vite :** Le projet a été créé avec npm create vite@latest mon-projet -- --template react.
* **Tailwind CSS & shadcn/ui :**
  + Installation de Tailwind CSS (npm install -D tailwindcss postcss autoprefixer @tailwindcss/postcss pour v4).
  + Configuration de tailwind.config.js et postcss.config.js.
  + Initialisation de shadcn/ui (npx shadcn@latest init), ce qui a configuré les variables de thème CSS, les alias d'importation (via vite.config.js), et créé components.json ainsi que src/lib/utils.js.
  + Ajout des composants shadcn/ui nécessaires (button, input, dialog, card, sonner, etc.) via npx shadcn@latest add <nom-composant>.
* **Structure des Composants :**
  + App.jsx : Sert de composant racine, gère l'état global (compte connecté, instance du contrat, liste des propriétés, etc.) et la logique principale d'interaction Web3.
  + Sidebar.jsx, Header.jsx : Composants de layout pour la structure visuelle du dashboard.
  + Des modales (Dialog shadcn/ui) ont été utilisées pour les formulaires d'ajout de propriété, de transfert, et pour afficher l'historique, améliorant l'UX.
  + Les propriétés sont affichées en utilisant des composants Card de shadcn/ui.
* **Interaction Web3 (ethers.js et MetaMask) :**
  + Connexion au portefeuille MetaMask (window.ethereum.request({ method: 'eth\_requestAccounts' })).
  + Instanciation du BrowserProvider et du Signer d'ethers.js.
  + Création d'une instance du contrat PropertyRegistry avec son adresse, ABI, et le Signer.
  + Appel des fonctions du contrat (lecture : getProperty, getTotalProperties, getPropertyHistory; écriture : addProperty, transferProperty).
  + Gestion des événements MetaMask (accountsChanged, chainChanged) pour la réactivité de la DApp.
* **Interaction IPFS (ipfs-http-client) :**
  + Initialisation du client IPFS pointant vers l'API du nœud local (http://localhost:5001).
  + Upload de multiples fichiers en tant que dossier avec ipfs.addAll(files, { wrapWithDirectory: true }) pour obtenir un CID de dossier unique.
  + Listage du contenu d'un dossier IPFS avec ipfs.ls(folderCid) pour la modale d'affichage des documents.
* **Feedback Utilisateur :** Le système de notifications sonner (intégré via shadcn/ui) a été utilisé pour fournir des retours clairs (succès, erreurs, informations) à l'utilisateur.

**8. Fonctionnalités Implémentées**

* Connexion/Déconnexion à MetaMask et affichage du compte.
* Distinction des rôles Administrateur (propriétaire du contrat) / Utilisateur standard.
* **Administrateur :** Formulaire d'ajout de nouvelle propriété (dans une modale) avec upload de multiples documents sur IPFS et enregistrement sur la blockchain.
* **Tous les utilisateurs :** Consultation de la liste des propriétés enregistrées, affichées sous forme de cartes.
* Affichage des détails de chaque propriété, y compris un lien vers le dossier des documents sur IPFS (ouverture d'une modale listant les fichiers du dossier).
* Affichage de l'historique des propriétaires pour chaque propriété (dans une modale).
* **Propriétaire d'un bien :** Possibilité de transférer la propriété à une autre adresse Ethereum via un formulaire (dans une modale).
* Interface utilisateur de type dashboard avec une sidebar pour la navigation entre les vues (Dashboard, Liste des propriétés) et un header.
* Notifications (toasts) pour les actions et les erreurs.

**9. Conclusion et Apprentissages**

Ce projet a permis de mettre en œuvre un cycle complet de développement d'une DApp, depuis la conception du smart contract jusqu'à la création d'une interface utilisateur interactive et moderne. Les principaux apprentissages incluent :

* La conception et l'écriture de smart contracts avec Solidity, en mettant l'accent sur la sécurité (contrôle d'accès, validations) et la gestion de l'état.
* L'utilisation de Hardhat comme environnement de développement Ethereum robuste.
* L'importance et la méthode d'intégration d'IPFS pour le stockage de données hors chaîne, en complément de la blockchain.
* Le développement frontend avec Vite et React, incluant la gestion d'état avec les Hooks.
* L'interaction avec la blockchain depuis React via ethers.js et MetaMask.
* L'intégration d'une bibliothèque de composants UI moderne comme shadcn/ui avec Tailwind CSS pour accélérer le développement d'interfaces de qualité.
* La gestion des opérations asynchrones et des retours utilisateur dans un contexte Web3.
* La résolution de problèmes liés à la configuration des outils (Tailwind v4, alias Vite, dépendances npm).

Ce projet constitue une base solide pour explorer des fonctionnalités plus avancées dans le domaine des DApps immobilières ou d'autres applications nécessitant traçabilité et décentralisation.

**10. Annexe : Commandes Clés Utilisées (Exemples)**

* **Backend (Hardhat) :**
  + npx hardhat compile
  + npx hardhat node
  + npx hardhat run scripts/deploy.js --network localhost
  + npm install @openzeppelin/contracts
* **Frontend (Vite & npm) :**
  + npm create vite@latest nom-du-projet -- --template react
  + npm install ethers ipfs-http-client
  + npm install -D tailwindcss@next postcss autoprefixer @tailwindcss/postcss (pour TWv4)
  + npx shadcn@latest init
  + npx shadcn@latest add <nom-composant> (ex: button, dialog, card, sonner)
  + npm run dev (pour lancer le serveur Vite)
* **Docker (pour IPFS) :**
  + docker-compose up -d