**架構圖**

+---------------------+ +----------------------------+ +-----------------------+

| | | | | |

| 前端應用 | <----------------- ---> | 後端服務 | <-----------> | 資料庫 |

| Reactjs | | Nodejs | | MongoDB |

| | | Express | | |

+---------------------+ +---------------------------+ +---------------------+

| |

| |

v v

+---------------------------------+ +----------------------------+

| | | |

| 使用者錢包 | | 以太坊區塊鏈 |

| MetaMask | | Smart Contracts |

+---------------------------------+ +----------------------------+

| |

| |

v v

+------------------------+ +-----------------------------+

| | | |

| 區塊鏈節點服務 | <-------------------> | 第三方服務 |

| Infura | | Infura |

+-------------------------+ +-----------------------------+

**組件說明**

1. **前端應用 (React.js)**：用戶界面，允許用戶與代幣進行互動，如查看餘額、轉帳等。
2. **後端服務 (Node.js/Express)**：處理業務邏輯、與智能合約交互、與資料庫通信。
3. **資料庫 (MongoDB)**：存儲用戶數據、交易記錄等。
4. **使用者錢包 (MetaMask)**：用戶的以太坊錢包，用於簽署交易和身份驗證。
5. **以太坊區塊鏈**：運行智能合約，管理代幣的發行和轉移。
6. **區塊鏈節點服務 (Infura)**：提供區塊鏈節點的訪問，避免自建節點的繁瑣。

**項目結構**

**BLOCKCHAIN**

**└─fjcu-token**

**├─artifacts**

**│ ├─@openzeppelin**

**│ │ └─contracts**

**│ │ ├─interfaces**

**│ │ │ └─draft-IERC6093.sol**

**│ │ ├─token**

**│ │ │ └─ERC20**

**│ │ │ ├─ERC20.sol**

**│ │ │ ├─extensions**

**│ │ │ │ └─IERC20Metadata.sol**

**│ │ │ └─IERC20.sol**

**│ │ └─utils**

**│ │ └─Context.sol**

**│ ├─build-info**

**│ └─contracts**

**│ └─FjcuToken.sol**

**├─backend**

**├─cache**

**├─contracts**

**├─frontend**

**└─scripts**

**規劃與需求分析**

* **代幣名稱**：fjcu
* **代幣總量**：21,000,000 枚
* **代幣標準**：ERC-20
* **功能需求**：
  + 發行代幣
  + 查詢代幣餘額
  + 轉帳代幣
  + 查看交易記錄

**設置開發環境**

**2.1 安裝必要工具**

* **Node.js 和 npm**：區塊鏈開發所需的JavaScript運行環境。
* **Git**：版本控制工具。
* **MetaMask**：瀏覽器擴展錢包，用於與前端應用互動。
* **IDE/編輯器**：如 VS Code。

**2.2 安裝智能合約開發工具**

* **Hardhat**：以太坊開發環境，用於編譯、部署和測試智能合約。

npm install --save-dev hardhat

* **Infura**：提供區塊鏈節點服務。

**註冊並獲取** [**Infura API Key**](https://infura.io/)

**3.1 初始化 Hardhat 專案**

mkdir fjcu-token

cd fjcu-token

npm init -y

npm install --save-dev hardhat

npx hardhat

選擇 **"Create a basic sample project"**，並按照提示完成初始化。

**3.2 撰寫 ERC-20 代幣合約**

在 contracts 目錄下創建 FjcuToken.sol 文件

**3.3 安裝 OpenZeppelin 套件**

npm install @openzeppelin/contracts

**3.4 編寫部署腳本**

在 scripts 目錄下創建 deploy.js 文件

**3.5 配置 Hardhat**

在 hardhat.config.js 中配置網絡

**3.6 部署智能合約**

npx hardhat compile

npx hardhat run scripts/deploy.js --network sepolia

部署成功後會在控制台看到合約地址。

佈署完可用測試網scan輸入合約地址查詢

**查詢測試網錢包餘額:**

npx hardhat run scripts/check-balance.js --network arbitrumSepolia

**查詢測試網區塊:**

使用以下命令，檢查網絡是否正確連接：

npx hardhat console --network sepolia

進入 Hardhat 控制台後，輸入以下命令來確認網絡連接：

const newprovider = ethers.getDefaultProvider();

newprovider.getBlockNumber().then((blockNumber) => {

console.log("Current block number:", blockNumber);

});

這會返回當前的區塊高度，如果能夠成功獲取到區塊高度，則說明 RPC 節點連接正常。如果出現問題，可能是 RPC 節點無法正常連接。

**4.1 初始化後端專案**

mkdir backend

cd backend

npm init -y

npm install express ethers mongoose dotenv cors

**4.2 配置環境變量**

在 backend 目錄下創建 .env 文件，內容如下：

PORT=5000

INFURA\_PROJECT\_ID=YOUR\_INFURA\_PROJECT\_ID

PRIVATE\_KEY=YOUR\_METAMASK\_PRIVATE\_KEY

CONTRACT\_ADDRESS=YOUR\_DEPLOYED\_CONTRACT\_ADDRESS

MONGODB\_URI=YOUR\_MONGODB\_CONNECTION\_STRING

替換 YOUR\_INFURA\_PROJECT\_ID、YOUR\_METAMASK\_PRIVATE\_KEY 和 YOUR\_DEPLOYED\_CONTRACT\_ADDRESS 為實際值。YOUR\_MONGODB\_CONNECTION\_STRING 是 MongoDB 的連接字符串，可以使用 [MongoDB Atlas](https://www.mongodb.com/cloud/atlas) 來免費創建。

**4.3 創建 Express 伺服器**

在 backend 目錄下創建 server.js 文件

**4.4 獲取智能合約的 ABI**

在 backend 目錄下創建 FjcuToken.json，並將 artifacts/contracts/FjcuToken.sol/FjcuToken.json 中的 ABI 複製進去。

在 Hardhat 部署後，ABI 會位於 artifacts/contracts/FjcuToken.sol/FjcuToken.json 文件中。將其中的 abi 部分複製到 backend/FjcuToken.json 中。

**4.5 啟動後端服務**

**運行 npm install**：在專案目錄下執行以下命令，將自動根據 package.json 安裝所有所需的依賴。

npm install

啟動服務命令

node server.js

如果一切配置正確，會看到 MongoDB connected 和 後端服務運行在 http://localhost:5000 的訊息。

**5.1 初始化前端專案**

**檢查 NPM 配置：**

執行以下命令來檢查 NPM 全局安裝目錄是否正確配置：

npm config get prefix

這會顯示你的 NPM 全局安裝目錄。如果路徑錯誤或不存在，則可以重新設置。

**設置正確的 NPM 全局路徑：**

如果路徑有問題，可以重新設置 NPM 的全局安裝目錄。執行以下命令將全局路徑設置到正確的位置，

npm config set prefix "C:\Users\ae887\AppData\Roaming\npm"

如果目錄不存在，手動創建該目錄。

**修復 NPM 安裝問題：**

嘗試修復 NPM 安裝路徑和文件：

npm install -g npm

**初始化**

npx create-react-app frontend

cd frontend

npm install ethers axios

**5.2 配置環境變量**

在 frontend 目錄下創建 .env 文件

REACT\_APP\_BACKEND\_URL=http://localhost:5000 REACT\_APP\_CONTRACT\_ADDRESS=YOUR\_DEPLOYED\_CONTRACT\_ADDRESS

將 YOUR\_DEPLOYED\_CONTRACT\_ADDRESS 替換為部署的智能合約地址。

**5.3 撰寫前端應用**

在 frontend/src 目錄下修改 App.js 文件

**5.4 獲取智能合約的 ABI**

在 frontend/src 目錄下創建 FjcuToken.json，並將智能合約的 ABI 複製進去。

與後端類似，將 Hardhat 編譯後 artifacts/contracts/FjcuToken.sol/FjcuToken.json 文件中的 abi 部分複製到 frontend/src/FjcuToken.json 中

**5.5 啟動前端應用**

npm start

前端應用將在 http://localhost:3000 運行。打開瀏覽器，連接 MetaMask 並與 DApp 互動。

**保護環境變量與版本控制**

**使用 .env 文件**：

確保環境變量只在後端的 .env 文件中配置，不要將 .env 文件提交到

Git。應在 .gitignore 文件中添加 .env，防止其被提交到版本控制系統。在 .gitignore 文件中添加： .env

**初始化 Git 儲存庫**： 在專案根目錄下執行以下命令來初始化 Git 儲存庫：

git init

**添加 .gitignore 文件**：

git add .gitignore

**將 .env 文件添加到 .gitignore 中**（如果還沒有的話）： 確保 .gitignore 文件中有以下條目：

.env

**設置遠程儲存庫，需要首先設置：**

git remote add origin <your-repo-url>

**創建並切換到 main 分支：**

如果當前沒有 main 分支，運行以下命令來創建並切換到 main 分支：

git checkout -b main

**推送到遠程儲存庫**：

git push origin main

**6.1 測試智能合約**

在 Hardhat 專案中撰寫單元測試，確保智能合約功能正確。

在 test 目錄下創建 FjcuToken.test.js

npx hardhat test

**使用 Postman 測試後端 API**

**獲取代幣餘額**

**請求方式**：GET

**URL**：http://localhost:5000/api/balance/0xYourWalletAddress

**Headers**：Content-Type: application/json

**保存 ERC-20 代幣轉帳記錄**

**請求方式：**POST

**URL：**http://localhost:5000/api/token-transfer

**Headers：**Content-Type: application/jso

**保存 ETH 轉帳記錄**

**請求方式：**POST

**URL：**http://localhost:5000/api/eth-transfer

**Headers：**Content-Type: application/json

**獲取所有交易記錄**

**請求方式：**GET

**URL：**http://localhost:5000/api/transactions

**Headers：**Content-Type: application/json

**連接 MongoDB Atlas（雲端資料庫）：**

mongosh "mongodb+srv://ae8877:ae8877@cluster0.rsmmk.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0"

**成功連接後，運行以下命令來查看所有資料庫：**

show dbs

**查看某個具體的資料庫：**

use <dbname>

**查看資料庫中的集合**

show collections

**查看集合中的數據**

db.<collection\_name>.find().pretty()

**過濾數據**

db.<collection\_name>.find({ name: "Alice" }).pretty()

**退出 MongoDB Shell**

exit

**6.2 部署到主網**

在主網部署智能合約會產生真實的 ETH 消耗。確保了解風險並已經做好準備。

**修改部署腳本**：

確保 hardhat.config.js 中配置了主網的 Infura URL 和私鑰。

**部署合約**：

npx hardhat run scripts/deploy.js --network arbitrum (使用二層arbitrum)

**更新後端和前端的合約地址**。

* 1. **部署前後端應用**

**後端部署**：

使用平台如 **Heroku**, **AWS (EC2, Elastic Beanstalk)**, **DigitalOcean** 等。

**前端部署**：

使用平台如 **Netlify**, **Vercel**, **GitHub Pages** 等。

**6.4 配置環境變量**

確保在部署後端和前端時，正確設置所有環境變量，如：

INFURA\_PROJECT\_ID

PRIVATE\_KEY

CONTRACT\_ADDRESS

MONGODB\_URI

REACT\_APP\_BACKEND\_URL

EACT\_APP\_CONTRACT\_ADDRESS

**7.1 智能合約安全**

**審計**：聘請專業的智能合約審計公司檢查合約代碼。

**使用標準庫**：利用 OpenZeppelin 的標準 ERC-20 實現，減少漏洞。

**測試覆蓋**：撰寫全面的單元測試和集成測試。

**7.2 應用安全**

**環境變量保護**：確保私鑰和敏感信息不被公開。

**API 安全**：實施認證和授權機制，防止未授權訪問。

**資料加密**：加密傳輸中的敏感數據，保護用戶隱私。

**7.3 性能優化**

**前端**：

減少不必要的區塊鏈請求。

使用緩存策略提高性能。

**後端**：

優化資料庫查詢。

使用緩存如 Redis 加速頻繁訪問的數據。

**7.4 監控與維護**

**監控智能合約**：

使用工具如 **Etherscan**, **Tenderly** 監控合約活動。

**應用監控**：

使用 **Prometheus**, **Grafana** 監控後端服務和資料庫性能。

**持續集成/持續部署 (CI/CD)**：

設置自動化測試和部署流程，提升開發效率和代碼質量。

**Arbitrum** 是目前最受歡迎的以太坊第二層解決方案之一，主要基於 **Rollup 技術** 來解決以太坊的可擴展性問題。以下是 **Arbitrum** 的特性、優勢與劣勢，以及其他可替代的解決方案。

**Arbitrum 的特性與優缺點**

**特性：**

1. **Rollup 技術**：Arbitrum 使用了「Optimistic Rollup」，這種技術將大量交易捆綁起來並提交到以太坊主鏈，只在必要時檢查交易的正確性，從而減少了計算和存儲成本。
2. **EVM 完全兼容**：Arbitrum 與以太坊虛擬機（EVM）完全兼容，因此開發者可以直接將現有的以太坊智能合約部署到 Arbitrum 上，幾乎不需要修改。
3. **安全性依賴於以太坊主網**：Arbitrum 通過以太坊的安全性來保護交易和資產，這使它在安全方面與以太坊主網幾乎一致。
4. **低 Gas 費**：相較於以太坊主網，Arbitrum 提供了更低的 Gas 費用，特別適合進行大規模交易和 dApp 部署。

**優點：**

* **降低交易成本**：Arbitrum 的 Optimistic Rollup 可以顯著減少交易成本，這對於需要頻繁交易的 dApp（如 DeFi 應用）特別有利。
* **高吞吐量**：Arbitrum 允許更快的交易處理速度，這使得整體網路效率提高。
* **強大的開發者支持**：Arbitrum 已經得到了許多開發工具和 dApp 生態系統的支持，並且有較多的資源和開發者社群。

**缺點：**

* **資金退出延遲**：Arbitrum 上的資金轉移回以太坊主網可能需要等待大約 **7 天**，因為 Rollup 需要在這段時間內進行糾紛解決，這對於需要快速流動性的用戶來說是一個挑戰。
* **集中化風險**：目前 Arbitrum 的開發和運行還由單一實體（Offchain Labs）控制，這在去中心化方面仍然有所限制。
* **不及 zk-Rollups 高效**：與 zk-Rollups（零知識 Rollup）技術相比，Optimistic Rollup 的驗證效率和安全性略遜一籌。

部署智能合約到以太坊主網的 Gas 費用 取決於多個因素，特別是以下幾個主要方面：

1. **智能合約的大小和複雜性：**合約越大或邏輯越複雜，部署所需的 Gas 也越多。每條指令、每個儲存操作都會消耗一定的 Gas。
2. **當前的 Gas Price（Gas 價格）：**Gas 費用是由 Gas 用量和當前的 Gas 價格決定的，Gas 價格是根據網路擁堵情況動態調整的。Gas 價格以 Gwei 為單位，1 Gwei 等於 0.000000001 ETH。
3. **當前的以太坊網絡擁堵狀況：**當網絡擁堵時，Gas 價格會變高，因為更多的用戶競爭同一區塊的空間。

**計算 Gas 費用**

* **Gas Limit（Gas 限制）：**這是部署合約的 Gas 上限，通常一個標準的 ERC-20 代幣智能合約的部署 Gas 限制在 200,000 到 500,000 Gas 之間，這是最常見的範圍。
* **Gas Price（Gas 價格）：**這是用來支付每單位 Gas 的價格。Gas 價格會根據網絡的擁堵情況浮動，通常在 10 到 100 Gwei 之間。

**計算公式：**

Gas 費用（ETH） = Gas 用量 \* Gas 價格

**估算費用**

假設部署一個標準的 ERC-20 代幣合約，Gas 用量可能在 200,000 到 500,000 Gas 之間。我們用這個範圍來估算 Gas 費用。

1. **當前 Gas 價格範例：**

* Gas 價格 = 20 Gwei
* 1 Gwei = 0.000000001 ETH

2. **計算範例：**

* 若合約需要 400,000 Gas：
  + Gas 費用 = 400,000 \* 20 Gwei = 8,000,000 Gwei = 0.008 ETH

所以，假設以太坊的價格為 1,500 美元，那麼：

* 0.008 ETH ≈ 12 美元

不同 Gas 價格情境下的範例：

| **Gas 用量** | **Gas 價格 (Gwei)** | **總 Gas 費用 (ETH)** | **費用 (USD) (假設 ETH = 1,500 美元)** |
| --- | --- | --- | --- |
| 200,000 | 20 Gwei | 0.004 ETH | 6 美元 |
| 400,000 | 20 Gwei | 0.008 ETH | 12 美元 |
| 500,000 | 50 Gwei | 0.025 ETH | 37.5 美元 |
| 500,000 | 100 Gwei | 0.05 ETH | 75 美元 |

**測試 RPC 節點連接**

手動測試 RPC 節點連接，確認是否能夠與網絡正常通信。

使用以下命令，檢查網絡是否正確連接：

npx hardhat console --network sepolia

進入 Hardhat 控制台後，輸入以下命令來確認網絡連接：

const provider = ethers.getDefaultProvider();

provider.getBlockNumber().then((blockNumber) => {

console.log("Current block number:", blockNumber);

});

**檢查餘額**

要檢查錢包中的 Sepolia 測試網 ETH 餘額，可以使用以下命令（替換 YOUR\_WALLET\_ADDRESS 為錢包地址）：

const walletAddress = "YOUR\_WALLET\_ADDRESS";

provider.getBalance(walletAddress).then((balance) => {

console.log("Balance:", ethers.utils.formatEther(balance), "ETH");

});

**部署腳本日誌**

如果網絡和餘額正常，可以繼續運行部署腳本。在執行部署腳本時，可以使用 --verbose 來查看詳細日誌：

npx hardhat run scripts/deploy.js --network sepolia --verbose