Architecture du Système de Gestion des Fournisseurs Basé sur la Blockchain

L'architecture du système repose sur une **approche modulaire** combinant une **blockchain (Ethereum ou Hyperledger)**, des **smart contracts**, un **backend API**, et une **interface web** pour l'interaction avec les utilisateurs.

---

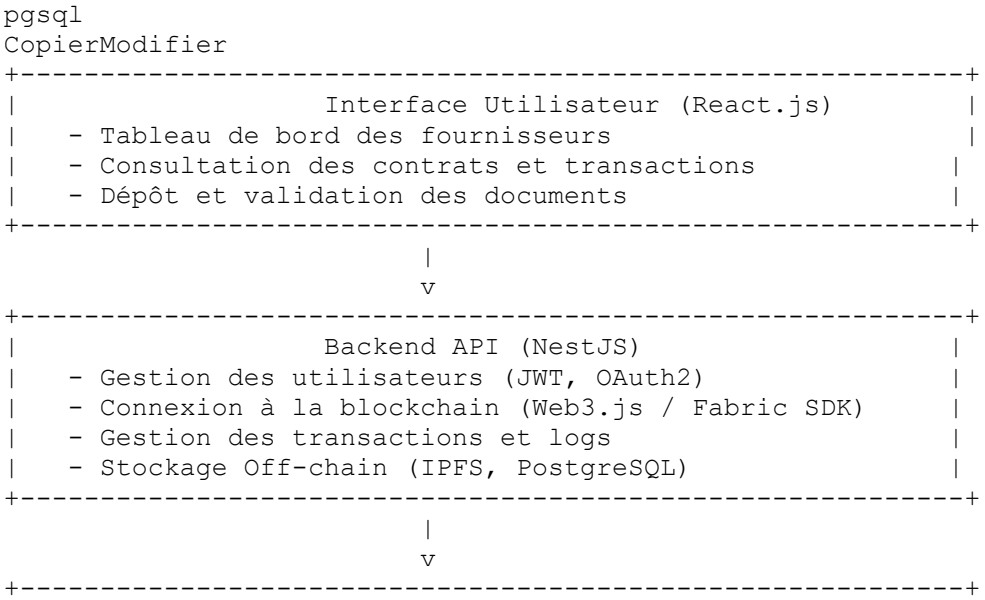
1. Vue d'Ensemble de l'Architecture

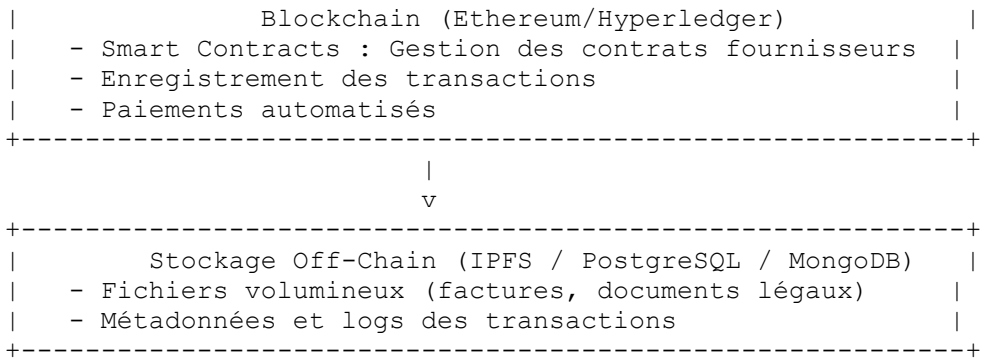
L'architecture est divisée en **quatre couches** principales :

1. **Interface utilisateur (Frontend)** :
   - Fournit un accès aux fournisseurs et aux entreprises.
   - Permet de consulter les contrats et suivre les transactions.
   - Utilisation de **React.js** pour une expérience utilisateur fluide.
2. **Backend API (Middleware)** :
   - Interagit avec la blockchain via Web3.js (Ethereum) ou Fabric SDK (Hyperledger).
   - Gère les autorisations et l'authentification des utilisateurs.
   - Stocke certaines informations hors blockchain (IPFS ou base SQL/NoSQL).
3. **Blockchain & Smart Contracts** :
   - Exécute et stocke les contrats intelligents.
   - Assure la traçabilité et la sécurité des transactions.
4. **Stockage Hors-Blockchain (Off-chain)** :
   - Utilisation de **IPFS** pour les fichiers volumineux (factures, documents).
   - Une base de données **PostgreSQL** ou **MongoDB** pour stocker les métadonnées.

---

2. Schéma de l'Architecture





---

3. Détails de l'Architecture

3.1. Interface Utilisateur (React.js / Next.js)

✅ Tableau de bord interactif pour les entreprises et fournisseurs.
✅ Suivi des contrats et paiements en temps réel.
✅ Connexion Web3 pour signer les transactions blockchain.

Exemple de connexion Web3 :

```
javascript
CopierModifier
import Web3 from "web3";

const web3 = new Web3(window.ethereum);
await window.ethereum.request({ method: "eth_requestAccounts" });
```

3.2. Backend API (NestJS + TypeORM)

✅ Communication avec la blockchain via Web3.js ou Hyperledger SDK.
✅ Gestion des utilisateurs et permissions via **JWT**.
✅ Enregistrement des logs dans PostgreSQL/MongoDB.

Exemple de connexion à Ethereum avec Web3.js :

```
typescript
CopierModifier
import Web3 from "web3";

const web3 = new Web3("https://mainnet.infura.io/v3/YOUR_INFURA_KEY");
const contract = new web3.eth.Contract(SmartContractABI,
"0xContractAddress");
```

3.3. Smart Contracts (Solidity / Hyperledger Chaincode)

✅ Gestion des fournisseurs et paiements sécurisés.
✅ Stockage des contrats et validation des livraisons.

Exemple de smart contract :

```
solidity
CopierModifier
pragma solidity ^0.8.0;

contract SupplierContract {
    address public buyer;
    address payable public supplier;
    uint public amount;
    bool public delivered;

    constructor(address payable _supplier, uint _amount) {
        buyer = msg.sender;
        supplier = _supplier;
        amount = _amount;
        delivered = false;
    }

    function confirmDelivery() public {
        require(msg.sender == buyer, "Only buyer can confirm");
        delivered = true;
    }

    function releasePayment() public {
        require(delivered, "Delivery not confirmed");
        supplier.transfer(amount);
    }
}
```

3.4. Base de Données Off-Chain (IPFS + PostgreSQL / MongoDB)

✅ **IPFS** : Stocke les documents (factures, certificats).
✅ **PostgreSQL / MongoDB** : Garde les métadonnées des transactions.

Exemple d'enregistrement d'un fichier sur IPFS avec Node.js :

```
javascript
CopierModifier
const ipfsClient = require("ipfs-http-client");
const ipfs = ipfsClient.create({ host: "ipfs.infura.io", port: 5001,
protocol: "https" });

async function uploadFile(fileBuffer) {
    const result = await ipfs.add(fileBuffer);
    console.log("IPFS Hash:", result.path);
}
```

---

4. Flux des Transactions

1️⃣ **Inscription du fournisseur** :

- Fournisseur soumet ses informations.
- Smart contract enregistre son identité sur la blockchain.

2️⃣ **Création d’un contrat fournisseur** :

- L’acheteur définit un contrat via un smart contract.
- Conditions stockées immuablement sur la blockchain.

3️⃣ **Validation de la livraison** :

- Le fournisseur soumet une preuve de livraison (hash IPFS).
- L’acheteur valide la livraison via un smart contract.

4️⃣ **Paiement automatique** :

- Smart contract libère les fonds si la livraison est validée.
- La transaction est enregistrée de manière transparente.

****

Plan progressif et structuré pour apprendre et développer le Système de gestion des fournisseurs basé sur la blockchain.

---

1. Comprendre les Fondamentaux de la Blockchain

Avant de coder, il est essentiel de comprendre comment fonctionne la blockchain.

À apprendre :

✅ Qu’est-ce qu’une blockchain ? (Décentralisation, sécurité, consensus)
✅ Différence entre **blockchain publique (Ethereum)** et **privée (Hyperledger Fabric)**
✅ Fonctionnement des **smart contracts** et **transactions**

---

2. Installer un Environnement de Développement Blockchain

Outils nécessaires :

✅ **Node.js** (Backend et Web3.js)
✅ **Ganache** (Blockchain Ethereum locale pour tester)
✅ **Truffle** ou **Hardhat** (Framework de développement Solidity)
✅ **Metamask** (Portefeuille pour signer les transactions)

👉 **Commandes pour l’installation** :

```
bash
```

```
CopierModifier
# Installer Node.js et vérifier la version
node -v

# Installer Truffle (facultatif, peut aussi utiliser Hardhat)
npm install -g truffle

# Installer Ganache CLI pour tester localement
npm install -g ganache-cli
```

---

3. Apprendre Solidity et Écrire son Premier Smart Contract

Solidity est le **langage de programmation des smart contracts** sur Ethereum.

Exemple simple d'un contrat pour enregistrer un fournisseur

```
solidity
CopierModifier
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SupplierRegistry {
    struct Supplier {
        string name;
        string email;
        bool isRegistered;
    }

    mapping(address => Supplier) public suppliers;

    function registerSupplier(string memory _name, string memory _email)
public {
        require(!suppliers[msg.sender].isRegistered, "Deja enregistre");
        suppliers[msg.sender] = Supplier(_name, _email, true);
    }

    function getSupplier(address _supplier) public view returns (string
memory, string memory) {
        require(suppliers[_supplier].isRegistered, "Non trouve");
        return (suppliers[_supplier].name, suppliers[_supplier].email);
    }
}
```

Tester ce contrat avec Hardhat

1 Initialiser un projet Hardhat

```
bash
CopierModifier
npx hardhat
```

2 Compiler le smart contract

```
bash
CopierModifier
npx hardhat compile
```

3️⃣ **Déployer localement avec un script JS**

---

4. Connecter le Smart Contract à une Interface Web

Pour interagir avec le smart contract depuis une page web, on utilise **Web3.js** ou **Ethers.js**.

Exemple d'interaction avec React.js

```
javascript
CopierModifier
import Web3 from "web3";
import SupplierRegistryABI from "./SupplierRegistryABI.json";

const web3 = new Web3(window.ethereum);
const contractAddress = "0x123..."; // Adresse du contrat déployé
const contract = new web3.eth.Contract(SupplierRegistryABI,
contractAddress);

async function registerSupplier(name, email) {
    const accounts = await web3.eth.getAccounts();
    await contract.methods.registerSupplier(name, email).send({ from:
accounts[0] });
}
```

---

5. Approfondir et Structurer le Projet

Maintenant que tu as les bases, voici comment structurer le projet :

Backend (NestJS + Web3.js)

✅ API REST pour gérer les transactions blockchain
✅ Connexion sécurisée avec JWT
✅ Stockage des données off-chain dans PostgreSQL

Frontend (React + Metamask)

✅ Interface utilisateur pour gérer les fournisseurs
✅ Interaction avec le smart contract via Web3.js

Blockchain (Ethereum ou Hyperledger Fabric)

✅ Déploiement sur un réseau testnet comme Rinkeby ou Goerli
✅ Gestion des paiements avec un contrat intelligent

---

6. Tester et Déployer le Projet

Test en local avec Ganache

```
bash
CopierModifier
ganache-cli
npx hardhat test
```

Déploiement sur un testnet Ethereum

```
bash
CopierModifier
npx hardhat run scripts/deploy.js --network goerli
```