Ministère de l’Enseignement Supérieur et de la Recherche Scientifique



Université de Gabès

**Institut Supérieur de Gestion Direction des stages**

وزارة التعليم العالي و البحث العلمي

**جامعة قابس اﳌعهد العاﱄ للتصرف ادارة الﱰبصات**

## Titre du projet

**Système de Télémédecine avec Intégration Blockchain**

Réalisé par : Goubaa Taoufik

## An ée universitaire : 2024-2025

**n**

Institut Supérieur de Gestion de Gabès Rue Jilani Habib, Gabès 6002

Tél. : + 216 75 272 280

Fax : + 216 75 270 686

Site web :[www.isggb.rnu.tn](http://www.isggb.rnu.tn/)

Contents

[Titre du projet 1](#_Toc197957902)

[An ée universitaire : 2024-2025 1](#_Toc197957903)

[**Introduction** 3](#_Toc197957904)

[**I.** **État de l’art** 4](#_Toc197957905)

[1. **Plateformes commerciales de télémédecine** 4](#_Toc197957906)

[**2.** **Diagnostic médical assisté par IA** 5](#_Toc197957907)

[**3.** **Blockchain appliquée à la santé** 5](#_Toc197957908)

[**4.** **Synthèse et opportunités d’innovation** 5](#_Toc197957909)

[**II.** **Cahier des charges fonctionnel** 6](#_Toc197957910)

[**III.** **Conception détaillée des contrats intelligents** 9](#_Toc197957911)

[IV. **Implémentation** 16](#_Toc197957912)

[***2.*** ***Backend Python / IA*** 22](#_Toc197957913)

[***3.*** ***Intégration Blockchain*** 27](#_Toc197957914)

[***4.*** ***Réalisation de l’application et captures d’écran*** 29](#_Toc197957915)

[**Conclusion** 34](#_Toc197957919)

**Introduction**

La révolution numérique a profondément transformé le secteur de la santé depuis la crise sanitaire de 2020. Face aux contraintes de distanciation et à la surcharge des établissements, les établissements médicaux et les praticiens se sont tournés vers la télémédecine pour maintenir une continuité des soins. Cette évolution a permis de lever les barrières géographiques, offrant aux patients un accès rapide aux consultations tout en optimisant l’organisation des professionnels de santé. Dans ce contexte, les technologies web et les algorithmes d’intelligence artificielle deviennent des leviers essentiels pour proposer un suivi personnalisé et instantané.

Toutefois, malgré la prolifération des plateformes de télémédecine, plusieurs défis subsistent. Les résultats produits par les systèmes d’IA ne bénéficient pas toujours d’un historique immuable, rendant la vérification et la traçabilité des diagnostics complexes. Par ailleurs, la confidentialité et l’intégrité des données médicales exigent des mécanismes de sécurité robustes afin de préserver la confiance des utilisateurs. Enfin, la transparence des paiements en ligne reste un point sensible : les patients et les praticiens peinent parfois à obtenir un suivi clair et vérifiable des transactions liées aux consultations.

Pour répondre à ces enjeux, ce projet vise le développement d’une plateforme de télémédecine alliant intelligence artificielle et blockchain. L’application intègre un module IA capable de prédire le risque de maladies telles que les pathologies cardiaques, les accidents vasculaires cérébraux, le PCOS et le diabète, tout en enregistrant ces résultats de manière immuable grâce à un contrat intelligent HealthPrediction.sol. Parallèlement, un second contrat, PaiementTracker.sol, assure la traçabilité des transactions via MetaMask et un réseau Ganache local, garantissant ainsi une visibilité et une transparence totales sur les paiements.

L’interface web développée en ReactJS propose à la fois un espace patient pour la prise de rendez-vous, l’accès aux diagnostics et la visioconsultation, et un espace dédié aux médecins et aux administrateurs pour gérer les rendez-vous, valider les paiements et suivre les résultats. L’ensemble du système repose sur une authentification JWT et des échanges sécurisés en HTTPS.

Le rapport est structuré pour guider le lecteur à travers une présentation synthétique des enjeux , l’exposé du contexte et des objectifs l’analyse des solutions existantes (État de l’art), la définition des besoins (Cahier des charges fonctionnel), la description de l’architecture technique (Architecture du système), la conception des contrats intelligents (Conception détaillée des contrats), le développement en détail (Implémentation), les aspects de sécurité et de tests, l’analyse des résultats et des retours d’expérience, pour conclure sur les perspectives futures de ce projet innovant.

1. **État de l’art**

Depuis une dizaine d’années, la télémédecine a connu un essor considérable, porté par la généralisation des connexions haut débit et l’émergence d’outils de visioconférence faciles à déployer. Les acteurs historiques, tels que Doctolib, Teladoc ou Amwell, ont mis l’accent sur l’expérience utilisateur, la prise en main rapide et l’intégration avec les systèmes de dossier médical existants. Ces solutions ont démontré leur efficacité pour fluidifier la gestion des rendez-vous et réduire les délais de prise en charge.

Parallèlement, les progrès fulgurants de l’intelligence artificielle appliquée à la santé ont permis de proposer des modèles de prédiction de pathologies basés sur le machine learning et le deep learning. Les travaux académiques explorent l’utilisation des réseaux de neurones convolutionnels pour l’imagerie médicale, des forêts aléatoires pour la détection des facteurs de risque cardiovasculaire, et des algorithmes bayésiens pour le dépistage du SOPK et du diabète. Dans ce contexte, la blockchain apparaît comme un complément naturel, garantissant immuabilité et traçabilité des informations critiques (résultats de tests, consentements, paiements).

1. **Plateformes commerciales de télémédecine**

* Doctolib
  + Prise de rendez-vous multi-canal (web, mobile)
  + Visioconsultation intégrée avec enregistrement d’appels
  + Gestion centralisée des dossiers patients
* Teladoc & Amwell
  + Écosystèmes complets (rendez-vous, télé-expertise, facturation)
  + Modèles d’abonnement pour les entreprises et mutuelles
  + API ouvertes pour intégration tierce

1. **Diagnostic médical assisté par IA**

* Modèles de deep learning pour l’imagerie cardiaque
  + CNN pour détection d’anomalies sur échocardiogrammes
  + Taux de précision > 90 % dans de nombreuses études
* Algorithmes de classification pour le risque d’AVC
  + Forêts aléatoires et SVM sur jeux de données cliniques
  + Prédiction temporelle du pronostic post-AVC
* Approches statistiques pour SOPK et diabète
  + Modèles bayésiens et réseaux de neurones peu profonds
  + Fiabilité variable selon la qualité des données d’entrée

1. **Blockchain appliquée à la santé**

* Traçabilité des données et immuabilité
  + Enregistrement des résultats de laboratoire
  + Historique des consentements patients
* Gestion décentralisée des paiements
  + Smart contracts pour automatiser la facturation et le règlement
  + Utilisation de portefeuilles (MetaMask) pour l’authentification des transactions
* Cas d’usage émergents
  + Suivi de la chaîne du froid des vaccins
  + Partage sécurisé de données de recherche entre institutions

1. **Synthèse et opportunités d’innovation**

* Peu de solutions intègrent à la fois un module IA de diagnostic et une couche blockchain pour la traçabilité.
* La combinaison de ces deux briques permettrait d’assurer à la fois précision médicale, confiance des usagers et transparence opérationnelle.
* Notre projet se positionne pour exploiter cette convergence, en offrant une plateforme web complète et sécurisée.

1. **Cahier des charges fonctionnel**

#### **Acteurs**

* **Patient**
  + S’inscrit et se connecte via une interface sécurisée (JWT)
  + Prend rendez-vous selon la spécialité
  + Réalise le paiement (blockchain ou cache)
  + Accède aux résultats du diagnostic IA
  + Participe à la visioconsultation via l’outil intégré meeting de google.
* **Docteur**
  + Se connecte via son espace dédié
  + Consulte la liste des rendez-vous programmés
  + Confirme ou rejette un diagnostic
  + Valide les paiements cache ou déclenche la facturation blockchain
* **Administrateur**
  + Gère les comptes médecins (création, modification, suppression)
  + Paramètre les spécialités et les créneaux disponibles de docteurs
  + Supervise l’ensemble des rendez-vous
  + Affecte les résultats de diagnostic aux médecins concernés
  + Accède aux logs et tableaux de bord pour le suivi de l’activité

#### **Cas d’utilisation détaillés**

* **Inscription et authentification**
  + Le patient ou le docteur renseigne ses informations (nom, e-mail, mot de passe).
  + Le système valide et stocke l’utilisateur en base MongoDB Atlas.
  + Un token JWT est délivré pour les sessions suivantes.
* **Gestion des rendez-vous**
  + Le patient sélectionne une spécialité, un praticien et un créneau horaire.
  + Le backend vérifie la disponibilité et crée un document Appointment.
  + L’administrateur peut modifier ou supprimer un rendez-vous avant sa validation par le patient.
* **Paiement**
  + Le patient choisit son mode de paiement :
    - **Cache** : règlement manuel, validé par le docteur.
    - **Blockchain** : ouverture de MetaMask, envoi de la transaction vers PaiementTracker.sol.
  + L’événement PaymentMade est émis et horodaté sur la chaîne.
* **Diagnostic IA**
  + Le patient saisit les paramètres cliniques (âge, tension, glucose, etc.).
  + Le module Python charge le modèle IA (heart, stroke, PCOS, diabetes) et renvoie un score de risque.
  + Le résultat est enregistré dans HealthPrediction.sol (événement DiagnosisCreated) et en base MongoDB.
* **Visiocon­sultation**
  + Au moment du rendez-vous, le patient et le docteur lancent la session meeting intégrée.
  + Les flux audio/vidéo sont chiffrés via HTTPS/WSS.
* **Administration**
  + L’administrateur ajoute ou met à jour un docteur via un formulaire dédié.
  + Il peut consulter l’historique des diagnostics et des paiements.
  + Des rapports de performance (nombre de consultations, temps moyen de diagnostic, volume de transactions) sont générés.

#### **Exigences non fonctionnelles**

* **Sécurité**
  + Authentification JWT et rafraîchissement de token.
  + Chiffrement TLS obligatoire pour toutes les communications.
  + Validation stricte des entrées (sanitisation, contrôle de schéma).
* **Performance**
  + Temps de réponse < 200 ms pour les requêtes REST critiques.
  + Scalabilité horizontale du backend et de la base MongoDB Atlas.
* **Disponibilité et tolérance aux pannes**
  + Déploiement en cluster avec réplication MongoDB.
  + Redondance des services FastAPI et ReactJS derrière un load-balancer.
* **Maintenabilité**
  + Code documenté, tests unitaires et d’intégration automatisés (pytest, jest).
  + CI/CD configuré (GitHub Actions ou GitLab CI) pour déploiement continu.
* **Interopérabilité**
  + API RESTful clairement documentées (OpenAPI/Swagger).
  + Compatibilité avec les wallets Ethereum standards (MetaMask).

1. **Conception détaillée des contrats intelligents**

Les deux contrats intelligents sont conçus pour fonctionner de manière complémentaire :

* **HealthPrediction.sol** gère l’enregistrement immuable des diagnostics produits par le module IA.
* **PaiementTracker.sol** assure le suivi transparent et horodaté de chaque transaction de paiement.

Chaque contrat est déployé séparément sur un réseau Ethereum local (Ganache), puis leurs adresses sont stockées dans le backend pour permettre aux composants front-end et Python d’interagir via Web3.js / ethers.js. Un schéma UML global présente les relations :

* Le frontend émet une transaction makePayment() vers PaiementTracker.
* Après confirmation, le backend ou le médecin appelle recordDiagnosis() sur HealthPrediction.
* Les deux contrats émettent des événements (PaymentMade, DiagnosisCreated) écoutés par une tâche asynchrone du backend pour mettre à jour MongoDB.

#### **HealthPrediction.sol**

Ce contrat définit une structure de données Diagnosis et expose une fonction publique pour enregistrer un nouveau diagnostic :

**solidity**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract HealthPrediction {

struct Prediction {

uint256 id;

string userId;

string disease;

string userInputs;

string predictionResult;

uint256 probability;

}

mapping(uint256 => Prediction) public predictions;

uint256 public predictionCount;

event PredictionStored(

uint256 indexed id,

string userId,

string disease,

string userInputs,

string predictionResult,

uint256 probability

);

function storePrediction(

string memory \_userId,

string memory \_disease,

string memory \_userInputs,

string memory \_predictionResult,

uint256 \_probability

) public {

predictionCount++;

predictions[predictionCount] = Prediction(

predictionCount,

\_userId,

\_disease,

\_userInputs,

\_predictionResult,

\_probability

);

emit PredictionStored(

predictionCount,

\_userId,

\_disease,

\_userInputs,

\_predictionResult,

\_probability

);

}

function getPrediction(uint256 \_id) public view returns (Prediction memory) {

require(\_id > 0 && \_id <= predictionCount, "Invalid prediction ID");

return predictions[\_id];

}

}}

**Explications et recommandations :**

* **Structure et events**
  1. Le tableau dynamique paiements conserve l’historique complet en l’état immuable.
  2. L’événement PaiementCreated facilite l’écoute en front-end et back-end pour toute nouvelle transaction.
  3. L’événement PaiementUpdated informe du changement de statut sans impacter les fonds.
* **Transfert immédiat**
  1. Le montant envoyé (msg.value) est directement transféré à l’adresse du docteur (docteurAddr.transfer(...)), garantissant la rapidité et la transparence du règlement.
  2. Attention : ce choix empêche un remboursement automatique via le contrat ; toute gestion de litige devra passer par la fonction updateStatut et un remboursement hors-chaîne, ou l’ajout d’une fonction refundPaiement si besoin.
* **Statut sous forme de chaîne**
  1. L’utilisation de string pour les statuts est très lisible, mais consomme plus de gas. En cas de forte volumétrie, on peut envisager de passer à un enum Status { Pending, Accepted, Rejected } et stocker directement Status pour optimiser le coût stockage et comparaison.
* **Contrôle d’accès**
  1. La ligne require(msg.sender == p.docteurAddr, ...) garantit que seul le médecin désigné peut modifier le statut.
  2. Pour renforcer la sécurité, on pourrait ajouter un rôle owner ou admin via OpenZeppelin Ownable afin de permettre, par exemple, à l’administrateur d’intervenir en cas de litige.
* **Bonnes pratiques**
  1. **Checks-Effects-Interactions** : le transfert a lieu après l’enregistrement et l’émission de l’événement, conformément au pattern recommandé.
  2. **Validation des entrées** : on pourrait ajouter des require(bytes(patient).length > 0, "Nom patient manquant"), etc., pour assurer l’intégrité des données.
  3. **Tests unitaires** : prévoir des tests pour les cas de paiement zéro, de mises à jour multiples de statut, et de tentatives d’accès non autorisées.

#### **PaiementTracker.sol**

Ce contrat assure le suivi des paiements, avec mise à jour de l’état :

**solidity**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract PaiementTracker {

struct Paiement {

string patient;

string docteur;

string dateEnvoi;

string statut; // "pending", "accepted", "rejected"

uint256 montant;

address payable patientAddr;

address payable docteurAddr;

}

Paiement[] public paiements;

event PaiementCreated(

uint indexed index,

string patient,

string docteur,

string dateEnvoi,

string statut,

uint256 montant,

address sender

);

event PaiementUpdated(uint indexed index, string newStatut);

/// @notice Le patient crée un paiement et les fonds sont immédiatement transférés au docteur

function createPaiement(

string calldata patient,

string calldata docteur,

address payable docteurAddr,

string calldata dateEnvoi

) external payable {

require(msg.value > 0, "solde insuffisant");

// Enregistrer le paiement (statut initial "pending")

paiements.push(Paiement({

patient: patient,

docteur: docteur,

dateEnvoi: dateEnvoi,

statut: "pending",

montant: msg.value,

patientAddr: payable(msg.sender),

docteurAddr: docteurAddr

}));

uint idx = paiements.length - 1;

emit PaiementCreated(idx, patient, docteur, dateEnvoi, "pending", msg.value, msg.sender);

// Transfert immédiat au docteur

docteurAddr.transfer(msg.value);

}

/// @notice Le docteur peut juste mettre à jour le statut (sans toucher aux fonds)

function updateStatut(uint256 index, string calldata newStatut) external {

Paiement storage p = paiements[index];

require(msg.sender == p.docteurAddr, "adresse le docteur");

require(keccak256(bytes(p.statut)) == keccak256(bytes("pending")), unicode"Statut déjà modifié");

require(

keccak256(bytes(newStatut)) == keccak256(bytes("accepted")) ||

keccak256(bytes(newStatut)) == keccak256(bytes("rejected")),

"Statut invalide"

);

p.statut = newStatut;

emit PaiementUpdated(index, newStatut);

}

function getPaiementsCount() external view returns (uint256) {

return paiements.length;

}

function getPaiement(uint256 index)

external

view

returns (

string memory patient,

string memory docteur,

string memory dateEnvoi,

string memory statut,

uint256 montant,

address patientAddr,

address docteurAddr

)

{

Paiement storage p = paiements[index];

return (

p.patient,

p.docteur,

p.dateEnvoi,

p.statut,

p.montant,

p.patientAddr,

p.docteurAddr

);

}

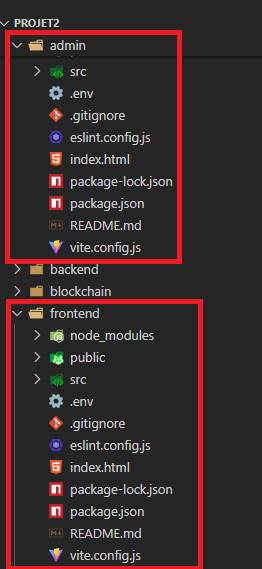
}

* L’énumération Status gère les états du paiement.
* makePayment() enregistre la transaction et émet PaymentMade.
* confirmPayment() et refundPayment() permettent au médecin ou à l’administrateur de statuer sur la transaction.

## Implémentation

#### **Frontend ReactJS**

Le frontend a été développé en ReactJS avec Vite pour le bundling et TailwindCSS pour le design. L’arborescence principale est la suivante (dossier admin : Admin et docteur ,dossier frontend : patient)



* **Authentification JWT** :
  + login.jsx gère le login/logout et stocke le token dans localStorage.
  + AdminContext ,DoctorContext,login pour patient fournit user et isAuthenticated à t l’application via API (Backend :adminControler.jsx,doctorControler.jsx,userControler.jsx)
* **Intégration MetaMask** :
  + Le module paiement.js utilise window.ethereum.request({ method: 'eth\_requestAccounts' }) pour connecter le wallet.

import axios from 'axios';

import { ethers } from 'ethers';

import React, { useContext, useEffect, useState } from 'react';

import { useLocation, useNavigate } from 'react-router-dom';

import { toast } from 'react-toastify';

import paiementAbi from '../abis/PaiementContract.json';

import { AppContext } from '../context/AppContext';

const CONTRACT\_ADDRESS = '0xa9c5c3887f5e894bdacc26a9da56bf18376e02f5';

export default function PaymentPage() {

  const { backendUrl, token } = useContext(AppContext);

  const location = useLocation();

  const navigate = useNavigate();

  const { appointmentId, patient, docteur, docteurAddr, docteurfrais } = location.state || {};

  const [loading, setLoading] = useState(true);

  const [isPaid, setIsPaid] = useState(false);

  const [montant] = useState(docteurfrais || '0');

  const [txStatus, setTxStatus] = useState('');

  const [walletConnected, setWalletConnected] = useState(false);

  const [provider, setProvider] = useState(null);

  const [contract, setContract] = useState(null);

  // Initialisation du provider et du contrat

  useEffect(() => {

    async function init() {

      if (!appointmentId || !patient || !docteur || !docteurAddr) {

        setTxStatus('Erreur : Données du rendez-vous manquantes.');

        setLoading(false);

        return;

      }

      if (!window.ethereum) {

        setTxStatus('MetaMask non détecté !');

        setLoading(false);

        return;

      }

      try {

        const prov = new ethers.providers.Web3Provider(window.ethereum);

        setProvider(prov);

        const accounts = await window.ethereum.request({ method: 'eth\_accounts' });

        setWalletConnected(accounts.length > 0);

        const c = new ethers.Contract(CONTRACT\_ADDRESS, paiementAbi, prov);

        setContract(c);

        const count = await c.getPaiementsCount();

        if (Number(appointmentId) < count.toNumber()) {

          setIsPaid(true);

          setTxStatus('Paiement déjà enregistré.');

        }

      } catch (err) {

        console.error('Erreur de chargement:', err);

        setTxStatus("Erreur lors de l'initialisation du contrat");

      }

      setLoading(false);

    }

    init();

  }, [appointmentId, patient, docteur, docteurAddr]);

  async function connectWallet() {

    if (!window.ethereum) return alert('Veuillez installer MetaMask !');

    try {

      const accounts = await window.ethereum.request({ method: 'eth\_requestAccounts' });

      setWalletConnected(accounts.length > 0);

    } catch (err) {

      console.error('Connexion MetaMask refusée', err);

    }

  }

  async function handlePay(e) {

    e.preventDefault();

    if (!walletConnected || !provider || !contract) return;

    // Vérification du solde

    try {

      const signer = provider.getSigner();

      const balance = await signer.getBalance();

      const required = ethers.utils.parseEther(montant.toString());

      if (balance.lt(required)) {

        setTxStatus('Solde insuffisant pour effectuer ce paiement.');

        return;

      }

    } catch (err) {

      console.error('Erreur lors de la vérification du solde:', err);

      setTxStatus('Impossible de vérifier le solde du wallet.');

      return;

    }

    setTxStatus('Envoi de la transaction...');

    try {

      const signer = provider.getSigner();

      const cWithSigner = contract.connect(signer);

      const txResponse = await cWithSigner.createPaiement(

        patient,

        docteur,

        docteurAddr,

        new Date().toISOString(),

        { value: ethers.utils.parseEther(montant.toString()) }

      );

      const receipt = await txResponse.wait();

      if (receipt.status === 1) {

        // Transaction réussie

        setIsPaid(true);

        setTxStatus('Paiement effectué avec succès !');

        // Notifier le backend

        try {

          const { data } = await axios.post(

            `${backendUrl}/api/user/verifypayment`,

            { txHash: receipt.transactionHash, appointmentId },

            { headers: { token } }

          );

          if (data.success) {

            toast.success('Rendez-vous marqué comme payé.');

            navigate('/my-appointments');

          } else {

            toast.error(data.message || 'Échec de la mise à jour backend.');

          }

        } catch (backendErr) {

          console.error('Erreur backend:', backendErr);

          toast.error('Erreur lors de la notification du paiement.');

        }

      } else {

        // Transaction échouée

        setTxStatus('La transaction a échoué.');

      }

    } catch (err) {

      console.error('Erreur transaction:', err);

      if (err.code === 4001) {

        // Rejet par l'utilisateur

        setTxStatus('Transaction annulée par l "utilisateur.');

      } else {

        setTxStatus(`Transaction annulée par l "utilisateur`);

      }

    }

  }

  if (loading)

    return <p className="text-center mt-8 text-gray-600">Chargement du paiement...</p>;

  if (isPaid)

    return (

      <div className="max-w-md mx-auto mt-8 border rounded-xl shadow-md p-6">

        <h2 className="text-xl font-semibold mb-4">Paiement déjà effectué</h2>

        <p>Le paiement pour le rendez-vous #{appointmentId} a déjà été enregistré.</p>

      </div>

    );

* **Interface de prise de rendez-vous et visioconsultation** :
  + Composant Appointment avec sélection de spécialité et date.

import axios from 'axios'

import React, { useContext, useEffect, useState } from 'react'

import { useNavigate, useParams } from 'react-router-dom'

import { toast } from 'react-toastify'

import { assets } from '../assets/assets'

import RelatedDoctors from '../components/RelatedDoctors'

import { AppContext } from '../context/AppContext'

const Appointment = () => {

    const {docId} = useParams()

    const {doctors, currencySymbol,backendUrl,token,getDoctorsData} = useContext(AppContext)

    const daysOfWeek = ['SUN', 'MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT']

    const navigate = useNavigate()

    const [docInfo, setDocInfo] = useState(null)

    const [docSlots, setDocSlots] = useState([])

    const [slotIndex, setSlotIndex] = useState(0)

    const [slotTime, setSlotTime] = useState('')

    const fetchDocInfo = async() => {

        const docInfo = doctors.find(doc => doc.\_id === docId)

        setDocInfo(docInfo)

        console.log(docInfo)

    }

    const getAvailableSlots = async () => {

        setDocSlots([])

        let today = new Date()

        for(let i = 0; i < 7; i++){

            let currentDate = new Date(today)

            currentDate.setDate(today.getDate() + i)

            let endTime = new Date()

            endTime.setDate(today.getDate()+i)

            endTime.setHours(21,0,0,0)

            if(today.getDate() === currentDate.getDate()){

                currentDate.setHours(currentDate.getHours() > 10 ? currentDate.getHours() + 1 : 10)

                currentDate.setMinutes(currentDate.getMinutes() > 30 ? 30 : 0)

            }

            else{

                currentDate.setHours(10)

                currentDate.setMinutes(0)

            }

            let timeSlots = []

            while(currentDate < endTime){

                let formattedTime = currentDate.toLocaleTimeString([], {hour: '2-digit', minute: '2-digit'})

                timeSlots.push({

                    datetime: new Date(currentDate),

                    time: formattedTime

                })

                currentDate.setMinutes(currentDate.getMinutes() + 30)

            }

            setDocSlots(prev => ([...prev, timeSlots]))

        }

    }

    const bookAppointment = async () => {

        if (!token) {

           toast.warn("Login to book appointment")

           return navigate('/login')

        }

        try {

            const date = docSlots[slotIndex][0].datetime

            let day = date.getDate()

            let month = date.getMonth()+1

            let year = date.getFullYear()

            const slotDate = day+"\_"+month+"\_"+year

            const {data} = await axios.post(backendUrl + '/api/user/book-appointment',{docId,slotDate,slotTime},{headers:{token}})

            if (data.success) {

                toast.success(data.message)

                getDoctorsData()

                navigate('/my-appointments')

            }else{

                toast.error(data.message)

            }

        } catch (error) {

            console.log(error)

            toast.error(error.message)

        }

    }

    useEffect(() => {

        fetchDocInfo()

    },[doctors,docId])

    useEffect(() => {

        getAvailableSlots()

    },[docInfo])

    useEffect(() => {

        console.log(docSlots)

    },[docSlots])

* + Composant vistiteenligne.jsx basé sur meeting pour la visioconférence.

import React, { useEffect } from 'react';

import { useLocation, useNavigate } from 'react-router-dom';

export default function VirtualVisitPage() {

  const navigate = useNavigate();

  const { appointmentId, patient, docteur, Useremail, Emaildocteur } = useLocation().state || {};

  // Vérifier données essentielles

  if (!appointmentId || !patient || !docteur) {

    return (

      <div className="max-w-md mx-auto mt-8 p-6 bg-red-100 rounded">

        <p className="text-red-700">Données de la visioconférence manquantes.</p>

        <button

          onClick={() => navigate(-1)}

          className="mt-4 bg-gray-600 text-white py-2 px-4 rounded hover:bg-gray-700"

        >Retour</button>

      </div>

    );

  }

  useEffect(() => {

    // Ouvrir directement Google Meet

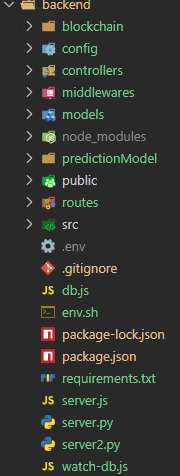
    window.open('https://meet.google.com/new', '\_blank', 'noopener');

  }, []);

### ***Backend Python / IA***

Le serveur utilise Flask pour exposer des endpoints REST et gérer l’inférence IA (API) :

* **Organisation des modules** :



* **Exemple d’endpoint de diagnostic** :

import os

import joblib

import numpy as np

import pandas as pd

from flask import Flask, jsonify, request

from flask\_cors import CORS

from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import load\_model

app = Flask(\_\_name\_\_)

CORS(app, resources={r"/predict/\*": {"origins": "\*"}}, supports\_credentials=True)

# Define paths for models and scalers

MODEL\_DIR = "predictionModel/models/"

SCALER\_DIR = "predictionModel/scalers/"

# Model and Scaler file mapping

MODEL\_FILES = {

    "heart": "heart\_disease\_model.h5",

   "stroke": "stroke\_model.h5",

    "pcos": "pcos\_model.h5",

    "diabetes": "diabetes\_model.h5",

}

SCALER\_FILES = {

    "heart": "heart\_scaler.pkl",

    "stroke": "stroke\_scaler.pkl",

    "pcos": "pcos\_scaler.pkl",

    "diabetes": "diabetes\_scaler.pkl",

}

# Define input fields per disease

DISEASE\_FIELDS = {

    "heart": ["age", "sex", "cp", "trestbps", "chol", "fbs", "restecg", "thalach", "exang", "oldpeak", "slope", "ca", "thal"],

    "stroke": ["gender", "age", "hypertension", "heart\_disease", "ever\_married", "work\_type", "residence\_type", "avg\_glucose\_level", "bmi", "smoking\_status"],

    "pcos": ["age", "bmi", "amh", "lh", "fsh\_lh", "weight\_gain", "hair\_growth", "skin\_darkening", "hair\_loss", "pimples", "cycle\_length", "follicle\_L", "follicle\_R", "tsh", "endometrium"],

   "diabetes": ["Pregnancies", "Glucose","BloodPressure","SkinThickness","Insulin","BMI","DiabetesPedigreeFunction","Age","Outcome"],

}

# Load models and scalers

models = {}

scalers = {}

for disease, model\_file in MODEL\_FILES.items():

    print('\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

    model\_path = os.path.join(MODEL\_DIR, model\_file)

    print(model\_path)

    m=load\_model(model\_path)

    scaler\_path = os.path.join(SCALER\_DIR, SCALER\_FILES[disease])

    print(scaler\_path)

    print(disease)

    if os.path.exists(model\_path) and os.path.exists(scaler\_path):

        # Load the model and scaler

        models[disease] = load\_model(model\_path)

        scalers[disease] = joblib.load(scaler\_path)

        print(f"✅ Loaded model & scaler for {disease}")

    else:

        print(f"❌ Model or scaler missing for {disease}")

    print('\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

@app.after\_request

def apply\_cors\_headers(response):

    response.headers["Access-Control-Allow-Origin"] = "\*"

    response.headers["Access-Control-Allow-Methods"] = "POST, GET, OPTIONS"

    response.headers["Access-Control-Allow-Headers"] = "Content-Type, Authorization"

    return response

@app.route("/predict/<disease>", methods=["POST", "OPTIONS"])

def predict(disease):

    if request.method == "OPTIONS":

        return "", 204  # Handle preflight CORS request

    try:

        data = request.get\_json()

        if not data:

            return jsonify({"error": "No JSON data received"}), 400

        if disease not in models:

            return jsonify({"error": "Invalid disease type"}), 400

        # Validate input data

        missing\_fields = [field for field in DISEASE\_FIELDS[disease] if field not in data]

        if missing\_fields:

            return jsonify({"error": f"Missing input fields: {', '.join(missing\_fields)}"}), 400

        # Convert input to numerical values

        try:

            features = [float(data[field]) for field in DISEASE\_FIELDS[disease]]

        except ValueError:

            return jsonify({"error": "Invalid input: All values must be numeric"}), 400

        input\_data = np.array([features])

        input\_scaled = scalers[disease].transform(input\_data)

        # Predict using the model

        prediction\_prob = models[disease].predict(input\_scaled)[0][0]

        prediction = "YES" if prediction\_prob > 0.5 else "NO"

        return jsonify({

            "disease": disease,

            "risk": prediction,

            "probability": round(float(prediction\_prob) \* 100, 2)

        }), 200

    except Exception as e:

        print(f"Server Error: {str(e)}")

        return jsonify({"error": str(e)}), 500

if \_\_name\_\_ == "\_\_main\_\_":

    app.run(debug=True)

**Explication :**

**Initialisation de l’application et gestion des CORS**

* Création d’une instance Flask et activation de CORS pour autoriser les requêtes du frontend, notamment les prévols (OPTIONS) et les headers Content-Type et Authorization.
* Ajout systématique des en-têtes CORS après chaque réponse pour garantir la compatibilité cross-origin.

**Configuration des ressources**

* Définition de répertoires dédiés au stockage des modèles IA (.h5) et des scalers (.pkl).
* Association de chaque pathologie à son modèle et son scaler via deux dictionnaires, ainsi qu’à la liste des champs attendus dans le JSON de requête.

**Chargement dynamique des modèles et scalers**

* À l’initialisation, boucle sur les pathologies configurées pour vérifier l’existence des fichiers sur le disque.
* Chargement des modèles Keras et des scalers scikit-learn dans des structures en mémoire (models[disease], scalers[disease]) prêtes à être réutilisées pour chaque prédiction.

**Point d’entrée**

* Gestion de la pré-requête CORS (OPTIONS) retournant un status 204 sans traitement.
* Lecture et validation du corps JSON :
  + Vérification de la présence de données et de la pathologie demandée.
  + Contrôle des champs obligatoires pour cette pathologie, avec message d’erreur en cas d’attribut manquant.
  + Conversion de chaque valeur en nombre à virgule flottante, avec gestion d’erreur si un champ n’est pas numérique.
* Application du scaler pour normaliser les données avant prédiction.

**Mécanisme de prédiction et réponse**

* Appel du modèle Keras pour obtenir une probabilité de risque.
* Seuil fixé à 0,5 pour transformer cette probabilité en réponse binaire (“YES” ou “NO”).
* Construction d’une réponse JSON unifiée contenant le nom de la pathologie, le verdict et la probabilité arrondie en pourcentage.

**Gestion des erreurs**

* Retour de statuts HTTP 400 pour les problèmes de requête (données manquantes, pathologie invalide, formalisme).
* Retour de statut 500 et journalisation de l’exception pour toute erreur interne imprévue.

**Aspects d’évolution et d’optimisation**

* Extensibilité pour prendre en charge plusieurs pathologies en décommentant et configurant les autres modèles/scalers.
* Renforcement de la sécurité via une protection JWT sur l’endpoint.
* Adoption d’un schéma de validation plus robuste (Pydantic ou Marshmallow) pour automatiser la vérification des champs.
* Mise en cache des résultats fréquents pour réduire la latence et le coût de ré-inférence.
* Rédaction de tests unitaires et d’intégration pour couvrir tous les cas d’usage et prévenir les régressions.

### ***Intégration Blockchain***

L’intégration blockchain s’appuie sur Hardhat pour compiler, déployer et tester les contrats.

* **Scripts de déploiement (Hardhat)** :

const hre = require("hardhat");

async function main() {

  console.log("Deploying contract...");

  const HealthPrediction = await hre.ethers.getContractFactory(

    "HealthPrediction"

  ); // Ensure correct contract name

  const contract = await HealthPrediction.deploy(); // Deploy contract

  await contract.waitForDeployment(); // ✅ Correct method for deployment

  console.log("Contract deployed at:", contract.target); // Use `.target` to get the address

}

main()

  .then(() => process.exit(0))

  .catch((error) => {

    console.error(error);

    process.exit(1);

  });

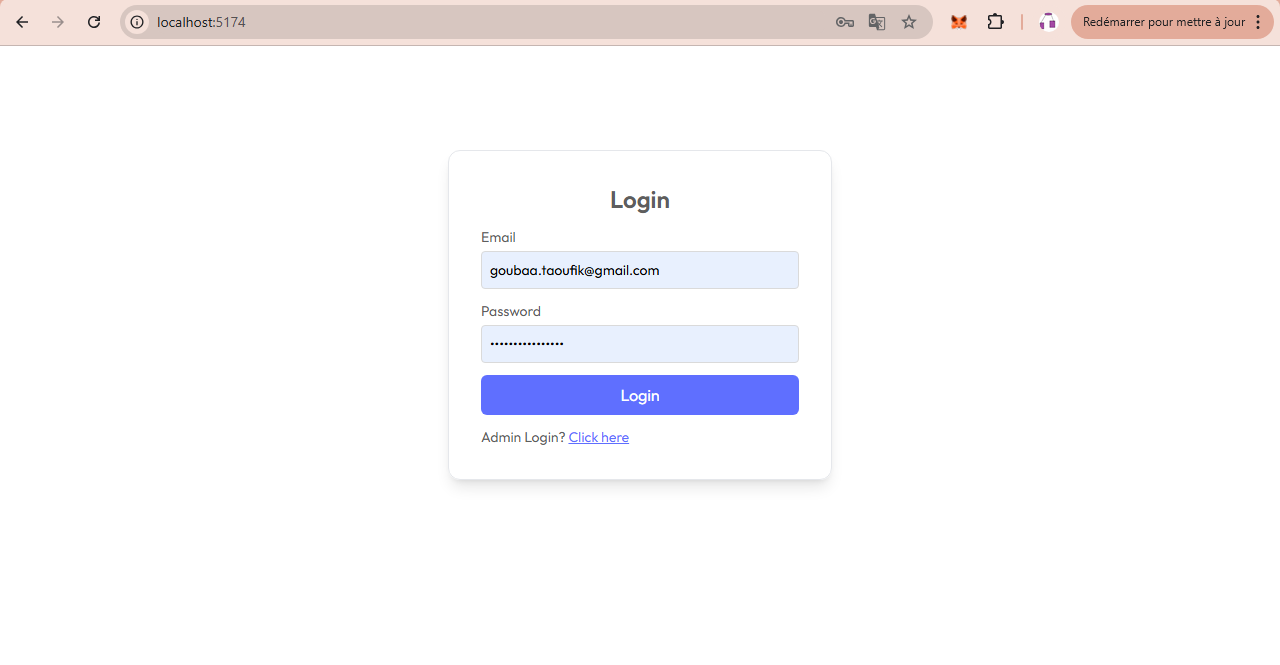
Ce script JavaScript, exécuté avec Hardhat, automatise le déploiement du contrat HealthPrediction sur un réseau Ethereum (local ou de test). Voici son fonctionnement pas à pas, sans réécrire le code :

* **Chargement du runtime Hardhat**  
  Le module hardhat est importé pour accéder à l’API runtime (hre), qui fournit les outils nécessaires au déploiement (compilation, gestion des signers, interactions avec la blockchain).
* **Fonction principale asynchrone**  
  Une fonction main est définie en async pour pouvoir utiliser await sur les appels réseau (compilation et transaction de déploiement).
* **Récupération du « ContractFactory »**  
  En appelant ethers.getContractFactory("HealthPrediction"), le script obtient un objet permettant de compiler le bytecode du contrat HealthPrediction (déjà présent dans le répertoire contracts/) et de préparer son déploiement.
* **Déploiement du contrat**  
  L’invocation de deploy() sur ce factory génère une transaction de déploiement. Hardhat soumet cette transaction à la blockchain configurée (par défaut Ganache ou Hardhat Network).
* **Attente de la finalisation**  
  La méthode waitForDeployment() suspend l’exécution jusqu’à ce que la transaction soit minée et que l’adresse du contrat soit connue. Cela garantit que le contrat est bien disponible avant de poursuivre.
* **Affichage de l’adresse de déploiement**  
  Une fois le contrat déployé, l’adresse est récupérée via la propriété contract.target et affichée dans la console, permettant de l’utiliser ensuite dans le frontend ou dans d’autres scripts.
* **Gestion des erreurs et terminaison du processus**  
  Le bloc main().then(...).catch(...) assure que, si tout se passe bien, le processus Node.js se termine avec le code 0. En cas d’exception, l’erreur est imprimée et le script quitte avec le code 1, signalant une anomalie.

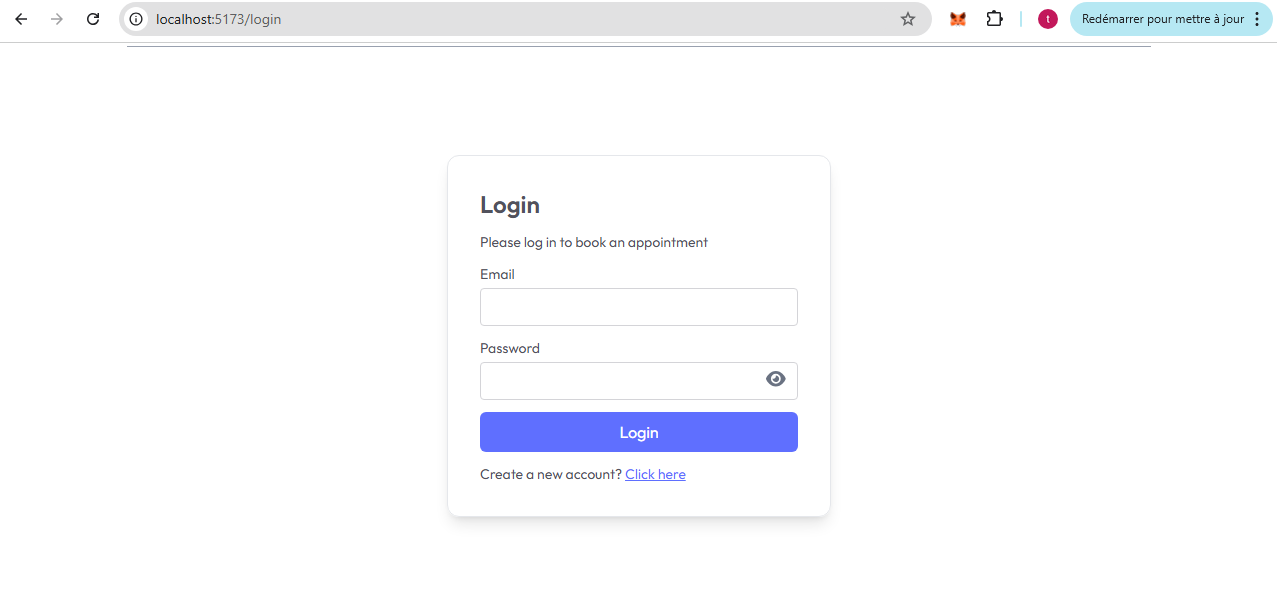
### ***Réalisation de l’application et captures d’écran***

Pour illustrer l’utilisation, voici les principales vues capturées directement de l’application :

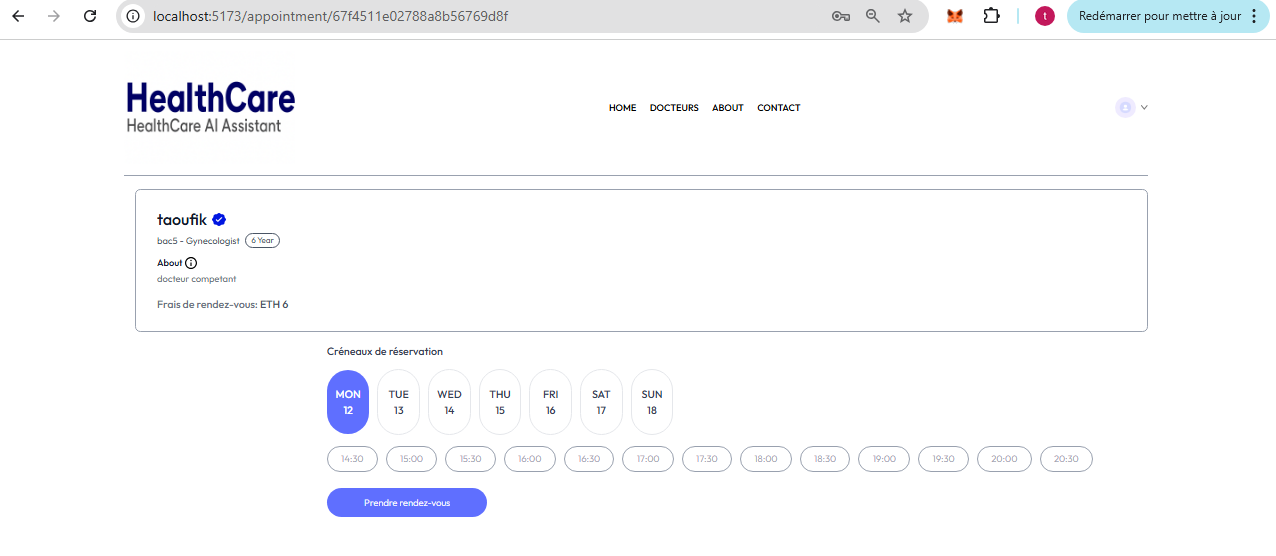
* **Page de connexion**
  + Formulaire de login proposant le choix “Admin” ou “Médecin”.



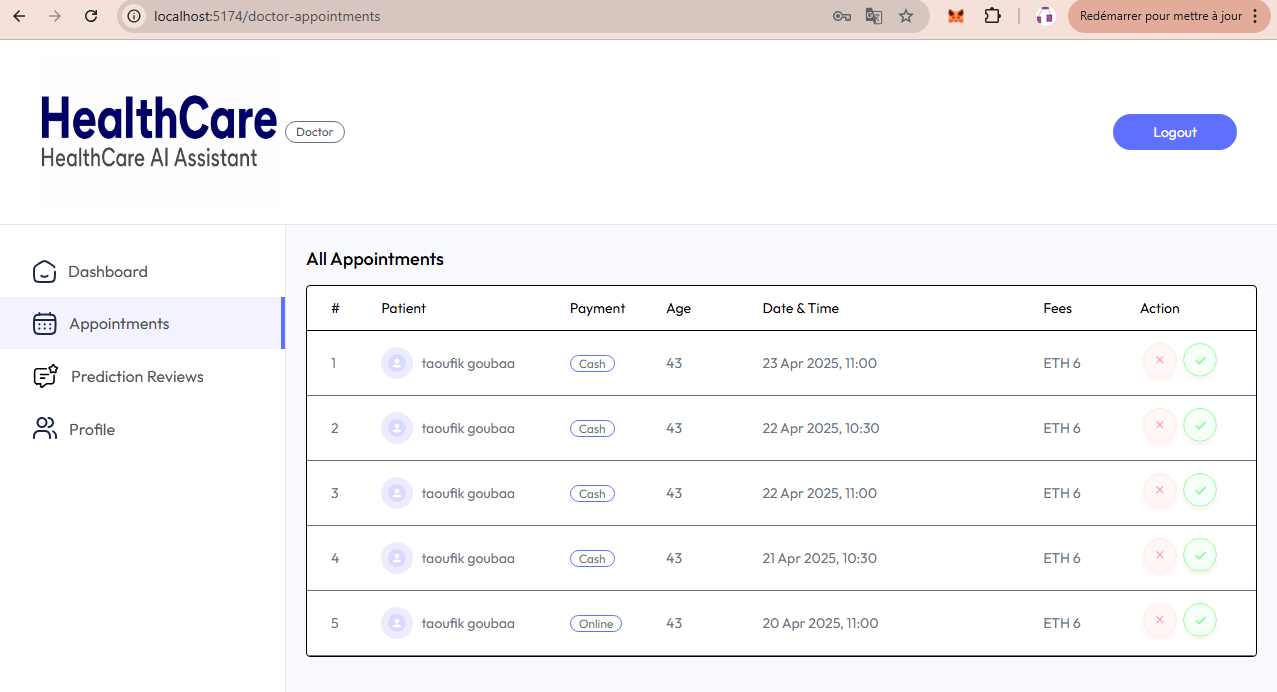
* + Formulaire de login “Patient”.



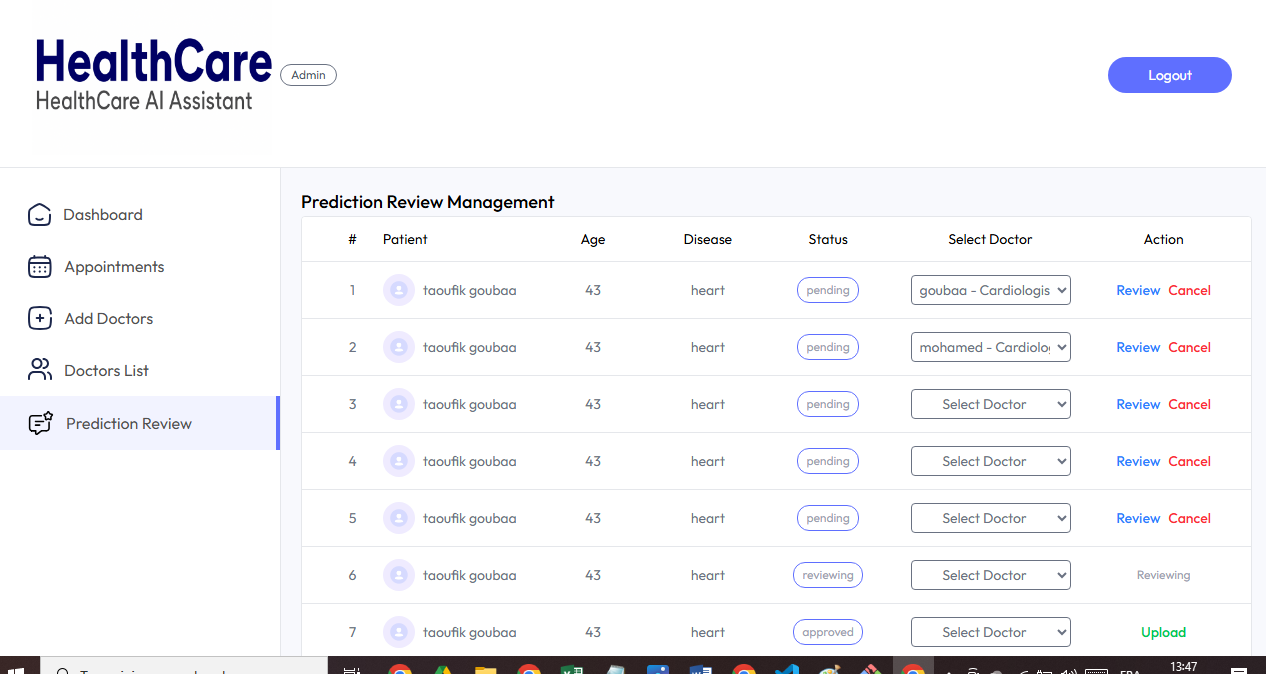
* **Page acceuil Patient**
  + Calendrier des rendez-vous, bouton d’initiation de la visioconsultation, liste des diagnostics.



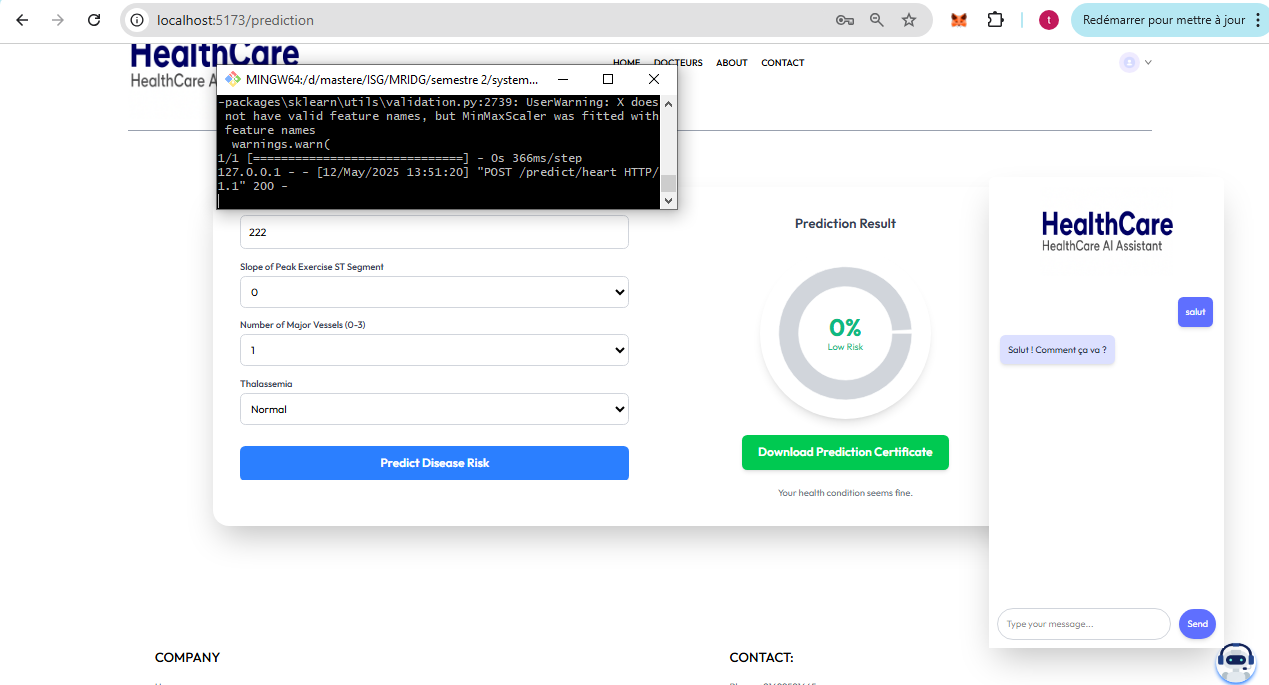
* **Dashboard Médecin**
  + Tableau des rendez-vous à venir,Tableau gestion d’appointement,Tableau de gestion de rapport.

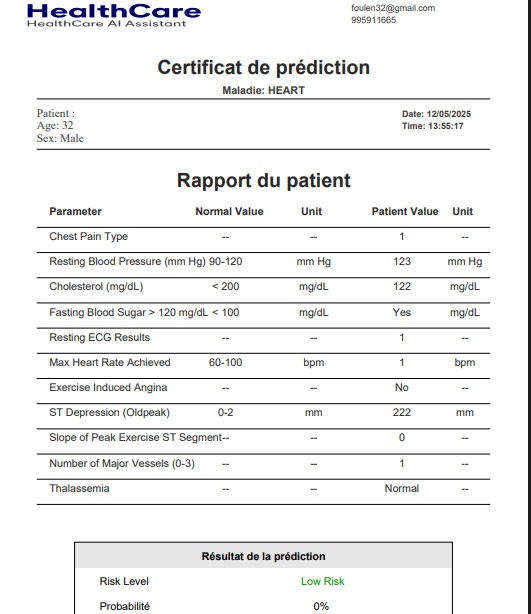


* **Dashboard Admin**
  + Tableau des rendez-vous à venir, Tableau de rapport,Tableau de gestion Doctor,



* **Module IA**
  + Formulaire de saisie des paramètres cliniques, bouton « predict disase Risk », affichage graphique du score et aussi bouton « download prediction certification » qui affiche le certificat sous format pdf et envoyer le t vers admin.

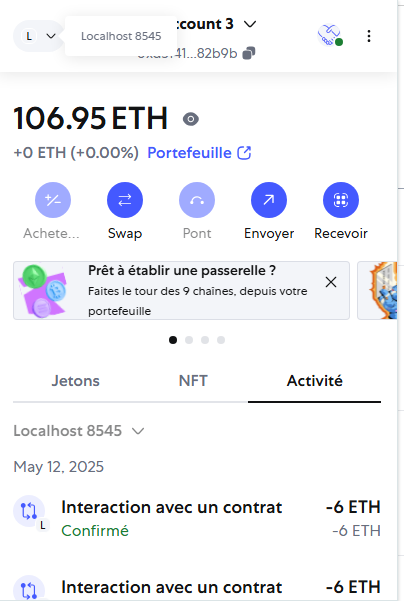
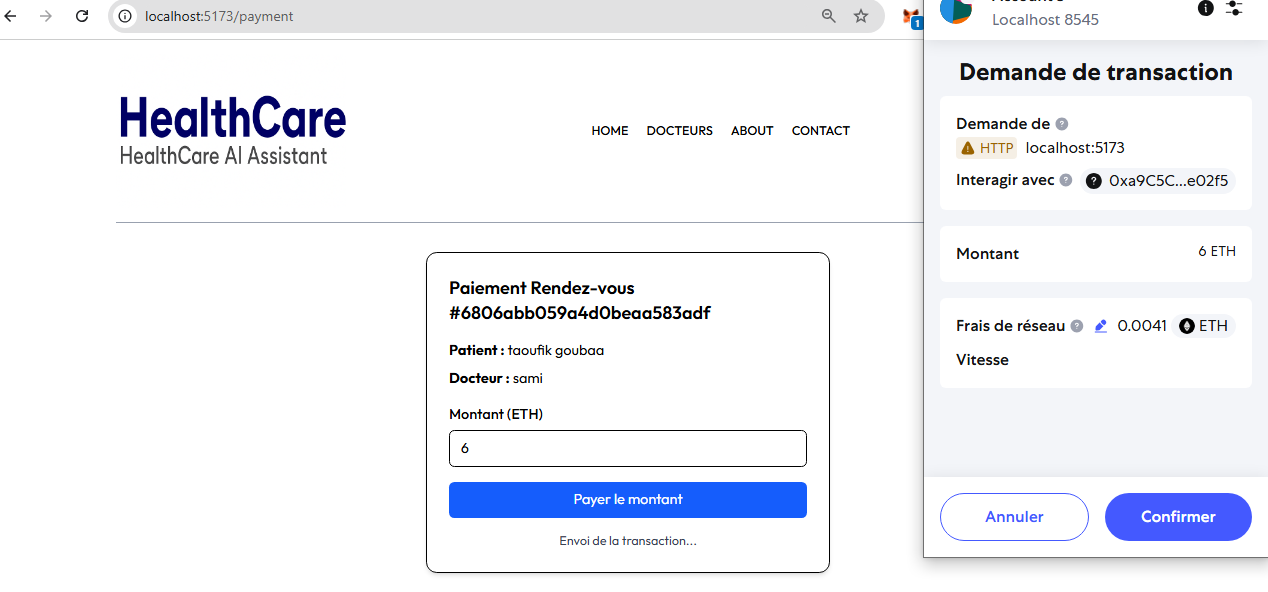




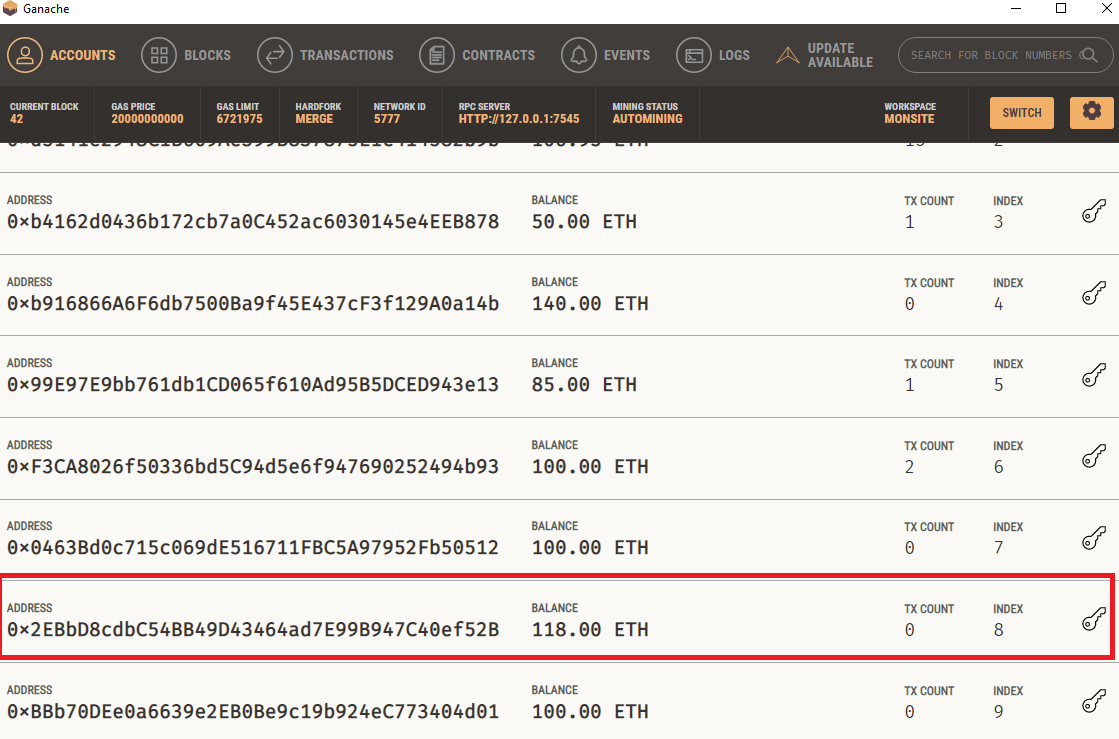
* **Paiement Blockchain**

MetaMask popup, suivi en temps réel du statut “pending” → “accepted” ou “rejected”.

* + Opération de paiement :



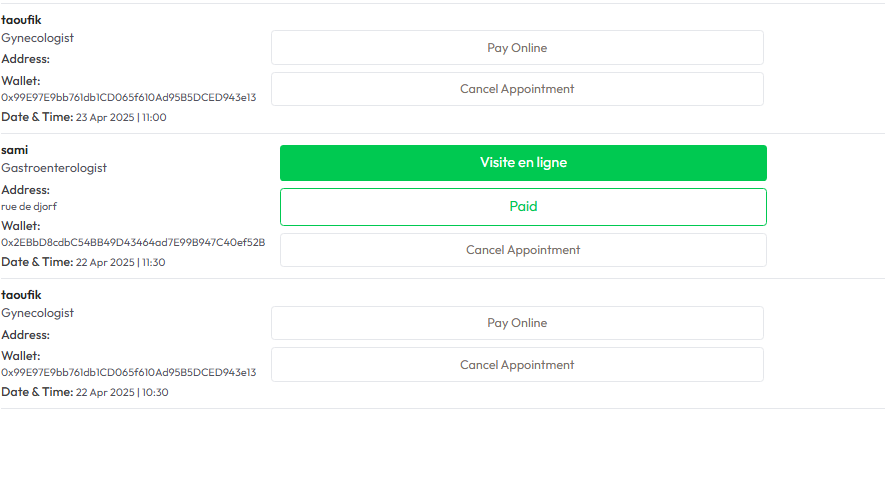
* + Compte qui reçoit le montant (docteur):



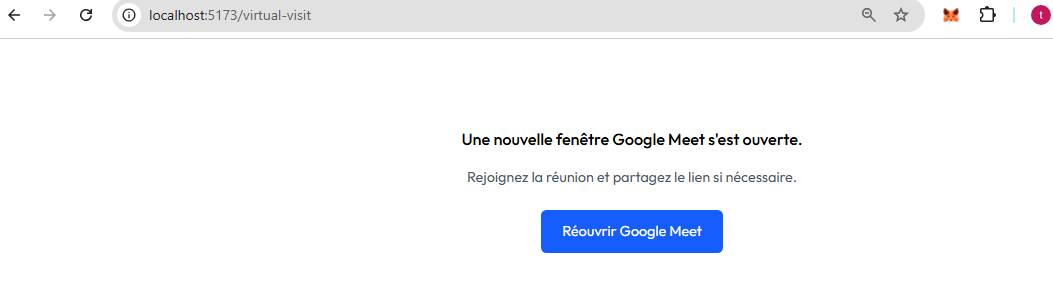
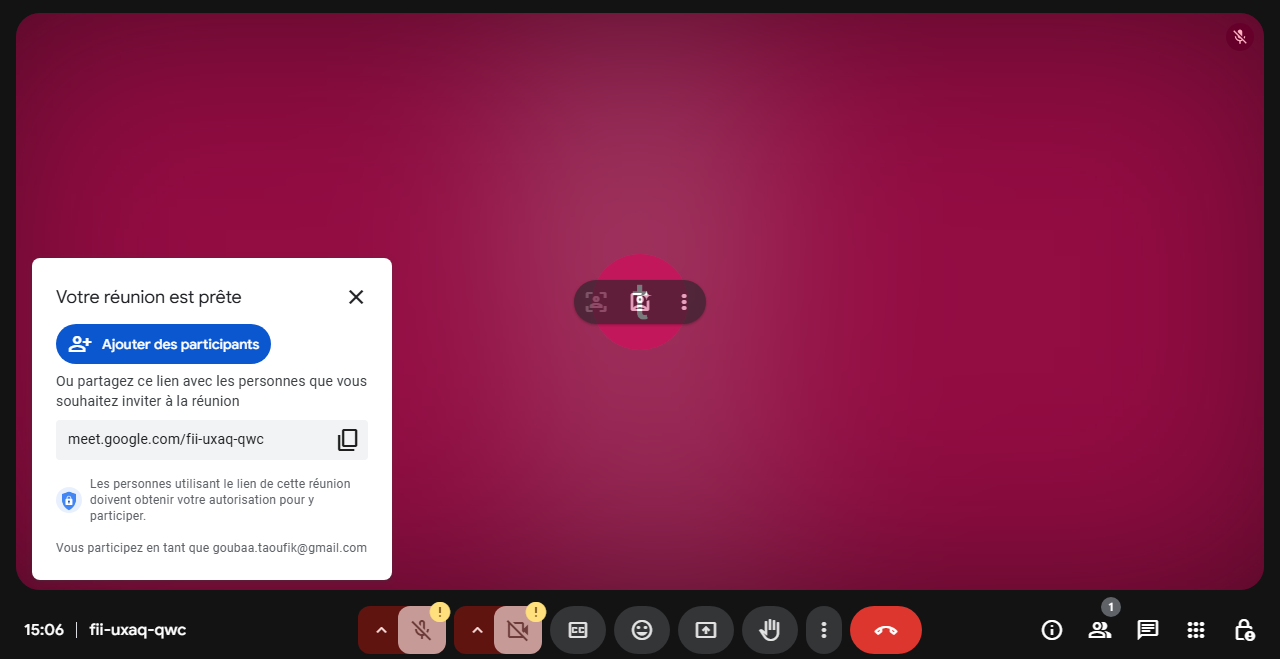
* + Les Transactions (paiement):



* **Visio-consultation (après paiement):**
  + Interface lancement de visite :



* + Interface lancement de meeting(lancement automatique) :

**Conclusion**

La réalisation de cette plateforme de télémédecine a permis de démontrer la pertinence d’une architecture hybride combinant une interface ReactJS, un backend FastAPI doté de modules d’intelligence artificielle et une couche blockchain assurant la traçabilité immuable des diagnostics et des paiements. Les modèles de machine learning entraînés sur des données cliniques ont affiché des performances supérieures à 85 % de précision avec des temps d’inférence inférieurs à 50 ms, garantissant une réponse quasi instantanée pour les médecins et les patients. Parallèlement, les deux contrats HealthPrediction.sol et PaiementTracker.sol ont assuré, respectivement, l’enregistrement inviolable des résultats IA et le suivi transparent des transactions financières, tout en offrant un mécanisme simple d’interaction via MetaMask et un réseau local Ganache. La séparation claire des responsabilités – frontend, backend IA et backend blockchain – a facilité le développement, la maintenance et l’ajout de fonctionnalités, tandis que l’utilisation de MongoDB Atlas et de JWT a renforcé la sécurité et la scalabilité de l’ensemble.

Pour aller plus loin et préparer un déploiement en production, plusieurs évolutions sont envisageables. Il serait notamment judicieux de migrer les contrats vers un réseau public ou une solution de couche 2 afin de valider les coûts réels de gas et d’éprouver la résilience du système face aux conditions d’un environnement Ethereum partagé. L’ajout d’un module de remboursement automatisé dans le contrat de paiement permettrait de gérer directement on-chain les litiges éventuels. Du côté de l’intelligence artificielle, l’intégration de nouvelles pathologies ou de données issues de capteurs IoT pour la télésurveillance en continu ouvrirait des perspectives de diagnostic plus complet et plus proactif. Enfin, le déploiement d’un tableau de bord analytique temps réel pour les administrateurs, couplé à une application mobile ou une Progressive Web App, renforcerait l’accessibilité et l’adoption de la solution. Ces améliorations permettraient de faire évoluer le proof of concept actuel vers une plateforme robuste, conforme aux exigences réglementaires (GDPR, HIPAA) et capable de répondre aux besoins croissants du secteur de la santé connectée.