Project on Blockchain Based HealthCare Solution

1. Title

HealthCareium

2. Project Members

Person 1

Name	Tahmid Rahman
Email	trahman5@buffalo.edu
Person Number:	50321139

Person 2

Name	Hamza Hafiz
Email	Hamzaha2@buffalo.edu
Person Number:	50478874

3. Issues Addressed

1. Secure management of electronic health records (EHRs)
2. Patient consent management
3. Data security
4. Insurance Transparency

5. Incentivization through secure payments.

4. Abstract

Blockchain technology can help healthcare experts and the overall healthcare industry to improve performance, patient data transparency, tracking, and accountability, as well as reduce costs.

Health data security is a top priority for all health systems and organizations, but the increasing volume of data is becoming a hurdle for providers to manage. Also, storing the data at a centralized location makes this sensitive data very vulnerable. By using Blockchain we can ensure that there is not a single gateway from which data can be tempered.

Currently in the United States, when we are new patients in a hospital, ER, or doctor's office, we need to fill up tons of forms about our health history and lifestyle. Sometimes, even if we are seeing the same doctor after a break of a few months, the first thing we do in the office is to fill up the same old form. Therefore, we can have a decentralized database of our medical history that can be shared in the medical department. It will save us lots of time and effort. Additionally, if a patient is in a critical condition in an emergency department. Then he/she will not have to hustle to share the medical history, instead, he/she will get the needed treatment quickly.

This application can be run by healthcare insurance companies in order to help experts and the overall healthcare industry to secure patient data more safely, track their medical history, as well as, to make the information more convenient to share. Health data security is a top priority for all health systems and organizations, but the increasing volume of data is becoming a hurdle for providers to manage. Also, storing the data at a centralized location makes this sensitive data very vulnerable. By using this application, we can ensure that there isn't a single gateway from which data can be tempered. We chose to build this application because keeping data transparent is our top priority unlike many insurance companies in the US. There were a number of cases recently where healthcare data has been leaked or shared unethically. In

fact, a lack of data transparency in healthcare leads to increased costs due to more errors, inaccurate diagnoses, and miscommunications.

5. How our application is Special

Since we already have health data of the patient we thought of leveraging it for other health industry use case which is health insurance. Insurance Transparency is a big issue in Insurance Sector. Patients claim the insurance companies charge excess premiums from them whereas Insurance companies claim that some patients forge their health test history & get health insurance at less premium.
Example: A patient who is suffering from chronic diseases can forge his health history and later claim the insurance money for treating that chronic disease. This should have been avoided at first place.

Since, Healthcareium is already storing the data of the patient in a secure manner d we have all the health history and health scores. So Healthcareium can be a single place of truth for patients as well as health Insurance agents.

1. For Patients , we specify based on his past health history what health condition category he falls into & based on that he gets to know what the right insurance premium for him is.
2. Simillarly , once health insurance agents have access to patient data , they can see whats the right insurance premium for the patient is and offer him insurance at that price.

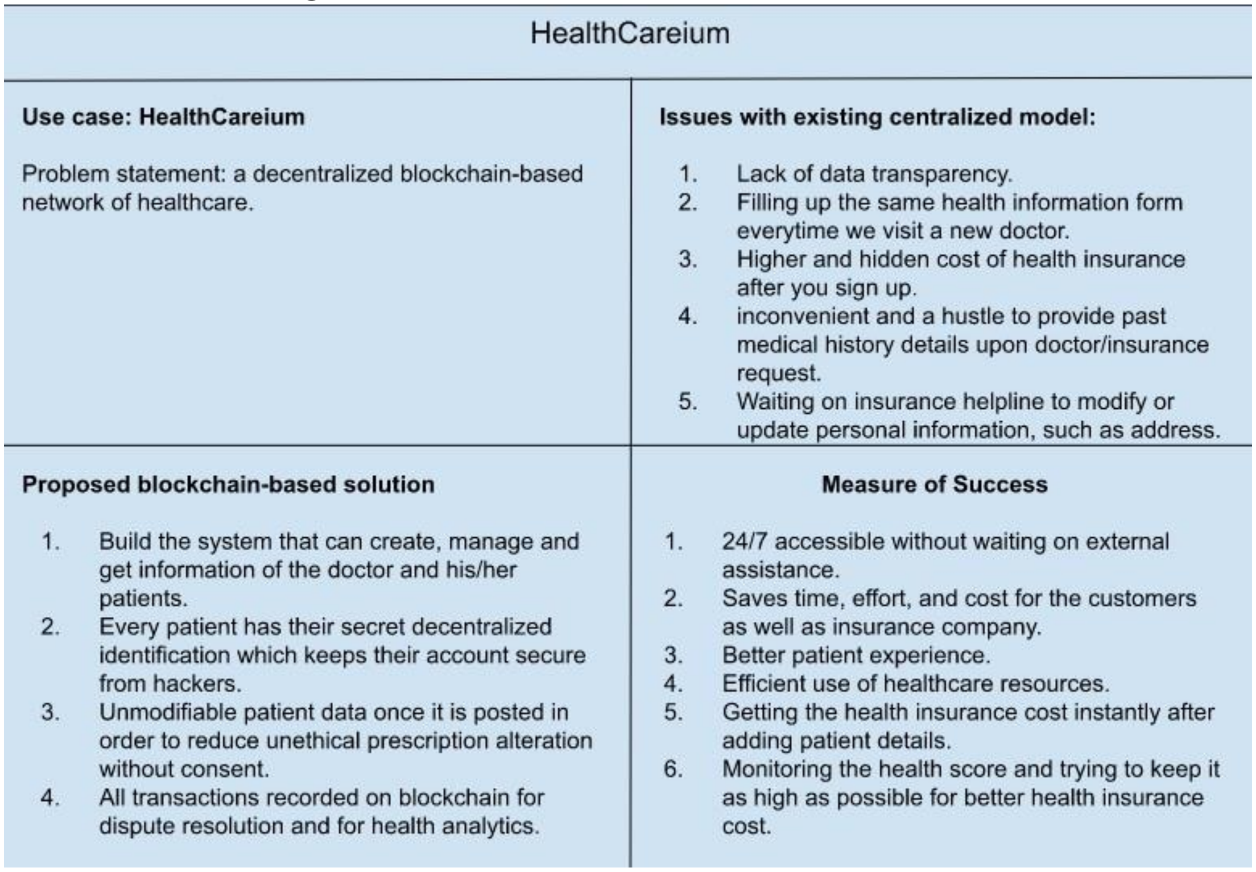
So no insurance frauds would occur anymore. We aim to serve both patients as well as insurance companies.

6. Approved By

Chen

7. Diagrams

Quad Chart Diagram:



HealthCareium

Use case: HealthCareium

Problem statement: a decentralized blockchain-based network of healthcare.

Issues with existing centralized model:

1. Lack of data transparency.
2. Filling up the same health information form everytime we visit a new doctor.
3. Higher and hidden cost of health insurance after you sign up.
4. inconvenient and a hustle to provide past medical history details upon doctor/insurance request.
5. Waiting on insurance helpline to modify or update personal information, such as address.

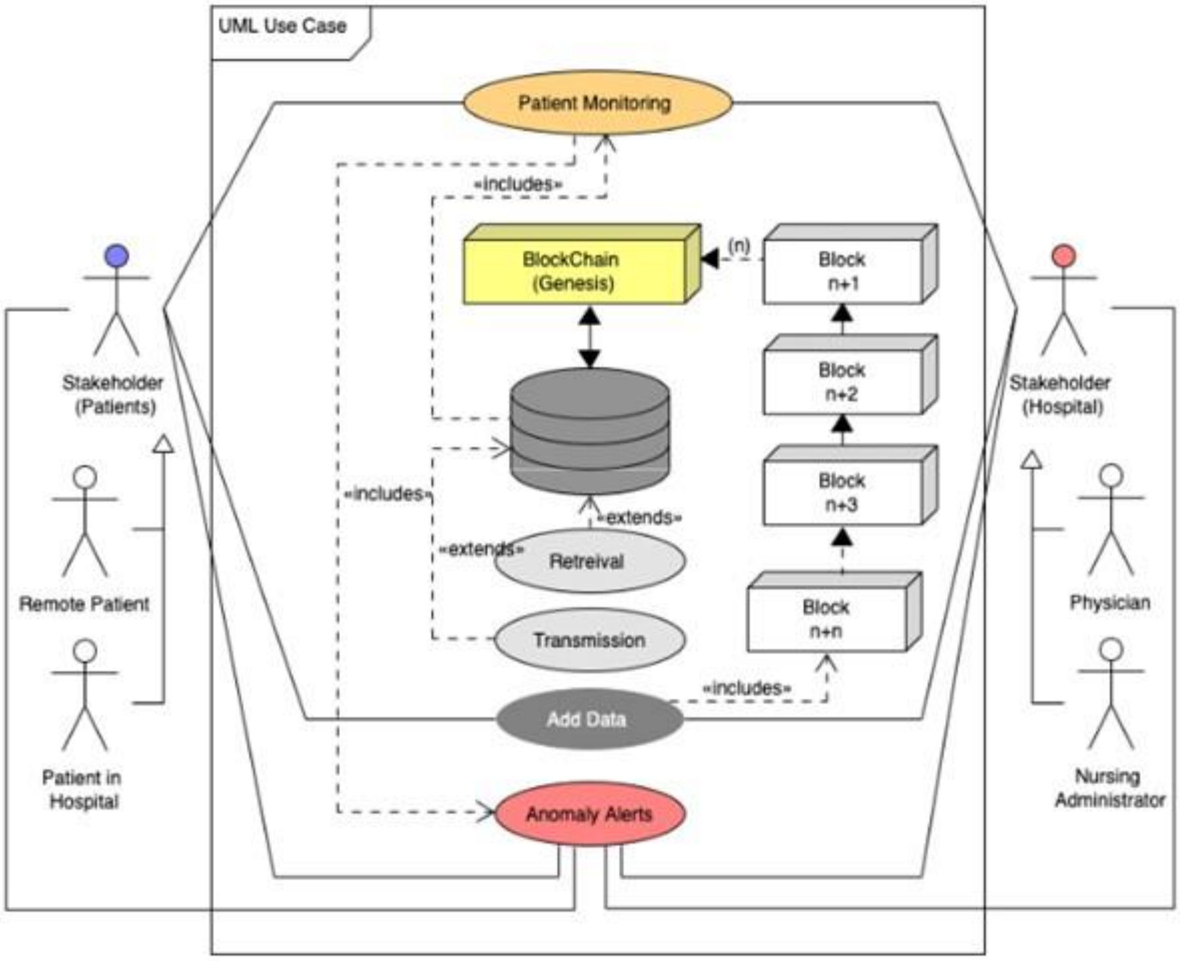
Proposed blockchain-based solution

1. Build the system that can create, manage and get information of the doctor and his/her patients.
2. Every patient has their secret decentralized identification which keeps their account secure from hackers.
3. Unmodifiable patient data once it is posted in order to reduce unethical prescription alteration without consent.
4. All transactions recorded on blockchain for dispute resolution and for health analytics.

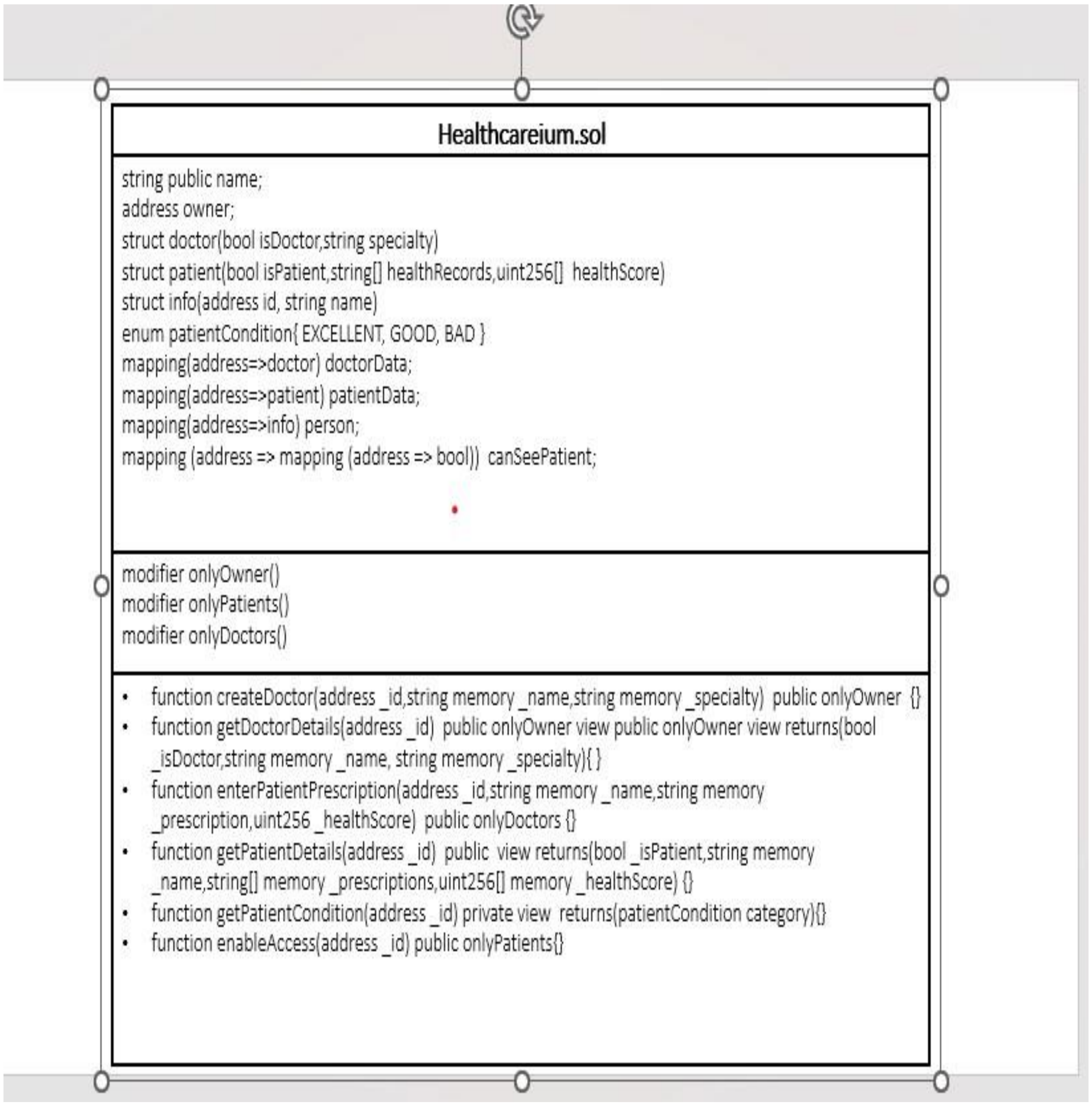
Measure of Success

1. 24/7 accessible without waiting on external assistance.
2. Saves time, effort, and cost for the customers as well as insurance company.
3. Better patient experience.
4. Efficient use of healthcare resources.
5. Getting the health insurance cost instantly after adding patient details.
6. Monitoring the health score and trying to keep it as high as possible for better health insurance cost.

Use Case Diagram:



Contract Diagram



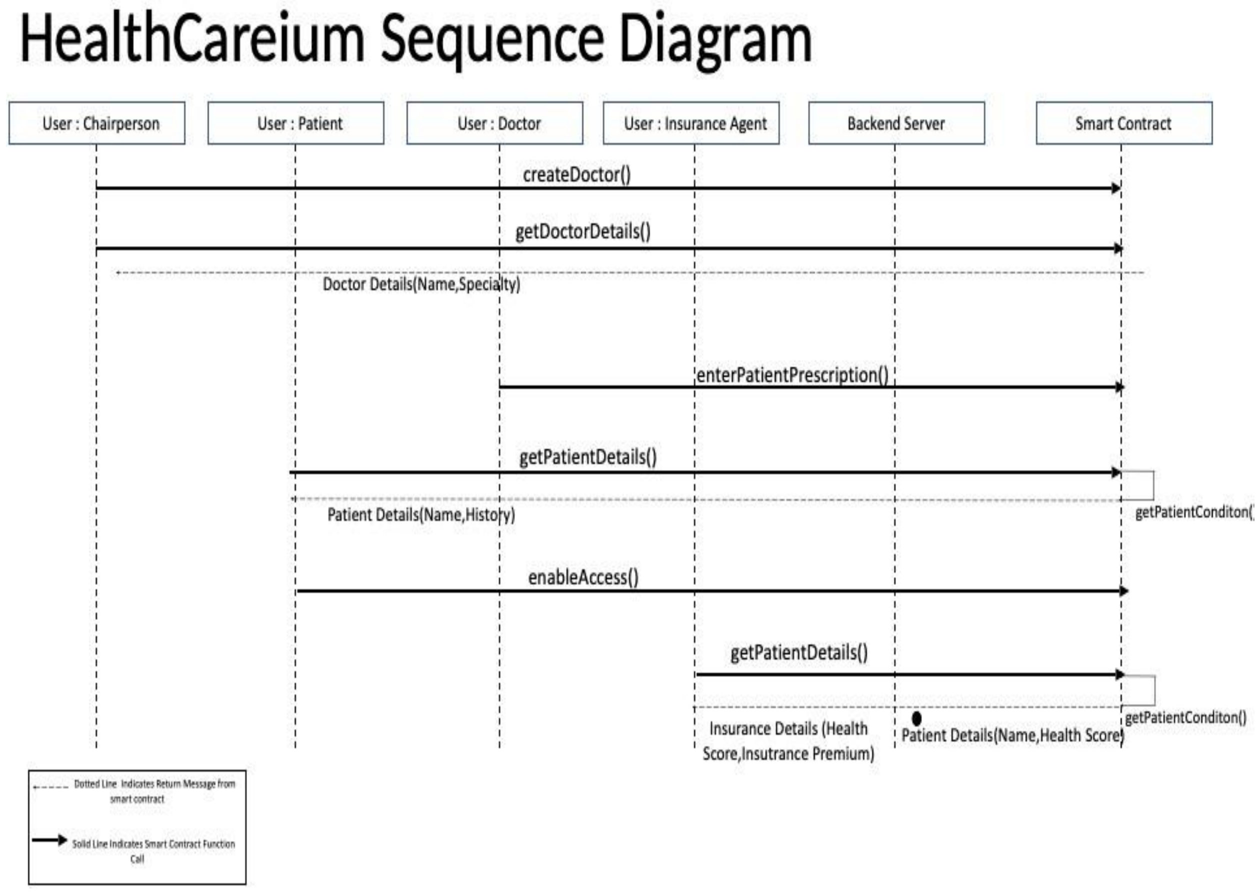
Healthcareium.sol

```
string public name;
address owner;
struct doctor(bool isDoctor,string specialty)
struct patient(bool isPatient,string[] healthRecords,uint256[]  healthScore)
struct info(address id, string name)
enum patientCondition{ EXCELLENT, GOOD, BAD }
mapping(address=>doctor) doctorData;
mapping(address=>patient) patientData;
mapping(address=>info) person;
mapping (address => mapping (address => bool))  canSeePatient;
```

```
modifier onlyOwner()
modifier onlyPatients()
modifier onlyDoctors()
```

- function createDoctor(address _id,string memory _name,string memory _specialty) public onlyOwner {}
- function getDoctorDetails(address _id) public onlyOwner view public onlyOwner view returns(bool _isDoctor,string memory _name, string memory _specialty){ }
- function enterPatientPrescription(address _id,string memory _name,string memory _prescription,uint256 _healthScore) public onlyDoctors {}
- function getPatientDetails(address _id) public view returns(bool _isPatient,string memory _name,string[] memory _prescriptions,uint256[] memory _healthScore) {}
- function getPatientCondition(address _id) private view returns(patientCondition category){}
- function enableAccess(address _id) public onlyPatients{}

Sequence Diagram:


HealthCareium Sequence Diagram
User : Chairperson
User : Patient
User : Doctor
User : Insurance Agent
Backend Server
Smart Contract
createDoctor()
getDoctorDetails()
Doctor Details(Name,Specialty)
enterPatientPrescription()
getPatientDetails()
Patient Details(Name,History)
getPatientConditon()
enableAccess()
getPatientDetails()
getPatientConditon()
Insurance Details (Health Score,Insutrance Premium)
Patient Details(Name,Health Score)
Dotted Line Indicates Return Message from smart contract
Solid Line Indicates Smart Contract Function Call

8. Instructions to Deploy,Interact & Test

Deploy:

RightNow in this phase we are deploying it to the local network because it will make it easy for testing it form multiple accounts. Please make sure your local blockchain ganache is running.
1. Go Into the folder where smart contract is.(HealthCareium Contract) 2. Run Terminal here.
3. Run Truffle Test(To check whether all test cases are passing)
4. Run Truffle Deploy
5. You should be able to see healthCareium smart contract address. Take it & paste in the contract variable in javascript file in the application.

Interact:

Our technology has 5 different pages with multiple functionalities. And those are explained below -

- **Home-** it is always nice to have a welcome page to acknowledge the customers because as we know a good first impression matters. When this page loads, the administrator will be redirected to the Metamask sign-in page(pop-up) and will be asked to log in.
- **Create Doctor-** This page is only accessible to the healthcare chairperson in our case the owner of the application(who deployed it) who can create doctors using their full name, their specialty, and a valid decentralized ID. Once those 3 forms are filled up, the information will be saved in the blockchain for the purpose of retrieving it later. ○ **Add Patient Details:** It will ask us to provide the patient's name, valid decentralized ID, prescription, and health score. The first two boxes are self-explanatory. But the prescription form saves the list of medication names assigned by the doctors. It is saved in the blockchain as well especially for the patient to retrieve it. Lastly, the health score is a number given by the doctor, which averages multiple health scores and provides an estimation of how much the health insurance will cost. We think this is a particularly useful feature that can give an idea to our customers right away about the approximate cost. As a result, the patients can be sure and confident in their decision of joining us. After all, our purpose of this technology is to help them by providing an honest and convenient service. ○ **Get Details:** This page provides all the information that was provided in the previous pages. To get the information, all three users - doctor, patient, and insurance will have to provide their valid names and decentralized IDs. Moreover, the information will be posted in white-boxed forms. The three forms are explained below-
    - **Doctor Details:** The doctor's information can be accessed only by the owner who deployed the contract. The person will have to provide the doctor's name and a valid decentralized ID.
    - **Patient Health Records:** This information is accessible by all three members- the insurance provider (to estimate the health care cost), the doctor (to monitor the patient's health), and of course the patient himself. We have created strong backend security which provides safety from outsiders by modifying the patient date without the valid member's consent.
  - **Insurance Details:** The information can be accessed by only the insurance provider if they need to monitor and verify their data. And, by the patient as well because they are the

main users who may want to keep track of the insurance information that is securely saved in the blockchain.

- **Manage Access:** This feature was added to modify or update all the information which were given in the last pages. Here only a patient can give access to other persons , if he wants them to see data.

Instructions to Test:

For convenience, the testing of the application can be easily done by deploying the smart contract on Ganache. As we would need multiple users to test the application & ensure privacy and security is ensured by checking data from the application is accessed only by people who have access to it.
Testing can be done using Truffle by running it against the unit test cases that we have written, the code for which can be seen below.

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


contract("healthCareium", (accounts) =>{
before(async () =>{
        instance = await healthCareium.deployed()
    })

    it('Validates an owner/admin  can create a doctor',async() =>{
result = await instance.createDoctor(accounts[1],"Dr.
James",{from : accounts[0]})
        //console.log(result.receipt.status)
        assert.equal(result.receipt.status,true,"Checking whwether
the admin can create doctor")



})
    it('Validates the doctor name is mapped correctly as
created',async() =>{         await
instance.createDoctor(accounts[1],"Dr. James",{from : accounts[0]})
        //console.log(result.receipt.status)

        let doctor = await
instance.getDoctorDetails(accounts[1],{from : accounts[0]})
assert.equal(doctor._isDoctor,true,"The doctor flag has been set to
true")         assert.equal(doctor._name,"Dr. James","The doctor
Name is also correctly mapped")

})
    it('Check if the doctor can create a patient',async() =>{
        result = await
instance.enterPatientPrescription(accounts[2],"Brian","Symptoms of
mild fever","7",{from : accounts[1]})
assert.equal(result.receipt.status,true,"Checking whether the doctor
can create doctor")

        //console.log(result.receipt.status)
    })
    it('Check whether the patient can get his details ',async() =>{
//console.log(result.receipt.status)

        let patient = await
instance.getPatientDetails(accounts[2],{from : accounts[2]})
       // console.log(patient)
        assert.equal(patient._isPatient,true,"The Patient flag has
been set to true")         assert.equal(patient._name,"Brian","The
Patient Name is also correctly mapped")

})

    it('Check whether the patient can manage his access',async() =>{
//console.log(result.receipt.status)
               await
instance.enableAccess(accounts[1],{from :
accounts[2]})

        let patient = await
instance.getPatientDetails(accounts[2],{from : accounts[1]})
       // console.log("The doctor is able to see details of
patient",patient)

        assert.equal(patient._isPatient,true,"The Patient flag has
been set to true")         assert.equal(patient._name,"Brian","The
Patient Name is also correctly mapped")

})

})
```

If an unauthorized user is accessing a page or form(method) he does not have access to , then a relevant error message will be displayed to him.

Steps: Below are stepwise guidelines for every page in our application **Create Doctor Test:**

1. In our application only the chairperson(the owner) who deployed the contract can create a doctor.
2. Login on Metamask using the address who deployed the contract.
3. Click on create doctor page.

4. Enter the relevant details.
5. Submit.
6. An alert will be generated if the doctor will be created. **Add Patient Details Test:**
    1. In our application only doctors can add patients details..
    2. Login on Metamask using the address of doctor.
    3. Click on ADD PATIENT DETAILS.
    4. Enter the relevant details.
    5. Submit.
    6. An alert will be generated if the Patient details will be successfully added. **Get Details Test:**
        1. In our application only persons who have access to data required can get those details.
        2. Login on Metamask using the address of the person.
        3. Fill in the relevant values in the form from which you want details.
        4. Only the Owner/Chairperson get patient details
        5. Only Patient or the person whom patient has given access can access can access details from GET PATIENT DETAILS & GET PATIENT INSURANCE.
        6. Enter the relevant details.
        7. Submit.
        8. An alert will be generated if the Patient details will be successfully added.

Manage Access:

1. Only Patient can give access to to his details by entering an address here.
2. Once patient gives access to any address they can access the patient details like patient history and health score.

9. Solidity Functions

The Smart Contract comprises of 5 functions which are mentioned below:

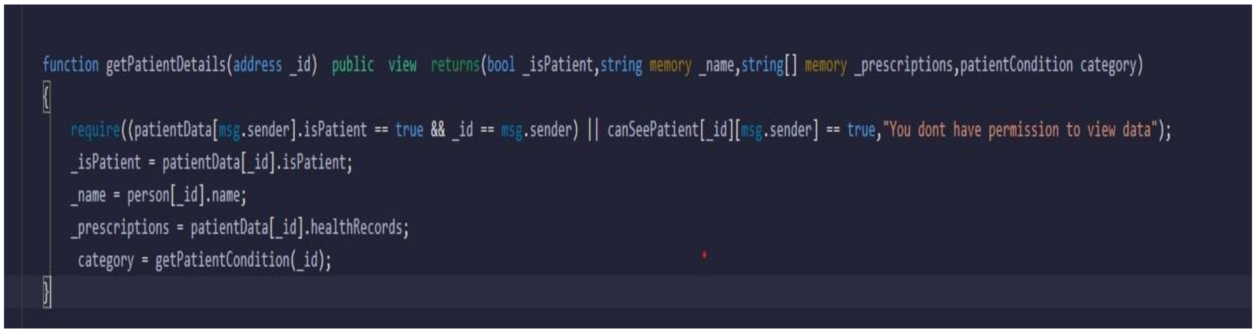
1. createDoctor
2. getDoctorDetails
3. enterPatientPrescription
4. getPatientDetails
5. enableAccess
6. getPatientCondition

Below we will investigate two smart contract functions in detail:

getPatientDetails :

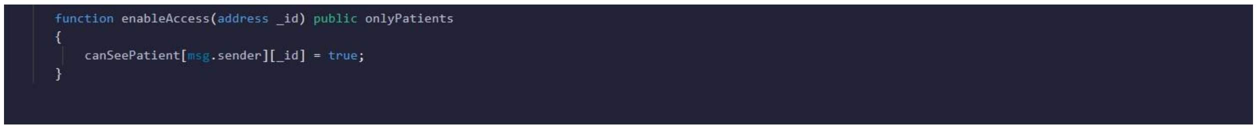
This method is responsible ensuring two use cases of the application:

a. Data Retrieval with focus on solving the data privacy issue
b. Insurance Premium Calculation to ensure Insurance Transparency. The method takes address of the patient for whom we need to calculate insurance or see the patient history. Require function here guarantees that only the person who have the permission should be able to execute this function



```
function getPatientDetails(address _id)  public  view  returns(bool _isPatient,string memory _name,string[] memory _prescriptions,patientCondition category)
{
    require((patientData[msg.sender].isPatient == true && _id == msg.sender) || canSeePatient[_id][msg.sender] == true,"You dont have permission to view data");
    _isPatient = patientData[_id].isPatient;
    _name = person[_id].name;
    _prescriptions = patientData[_id].healthRecords;
     category = getPatientCondition(_id);
}
```

The method can always be called by the patient for their data. Also , the patient can manage who else can see the their data(other people can be doctors, nurses ,Insurance Agents). The access to the patient data can be managed by enableAcess Method which is shown below:

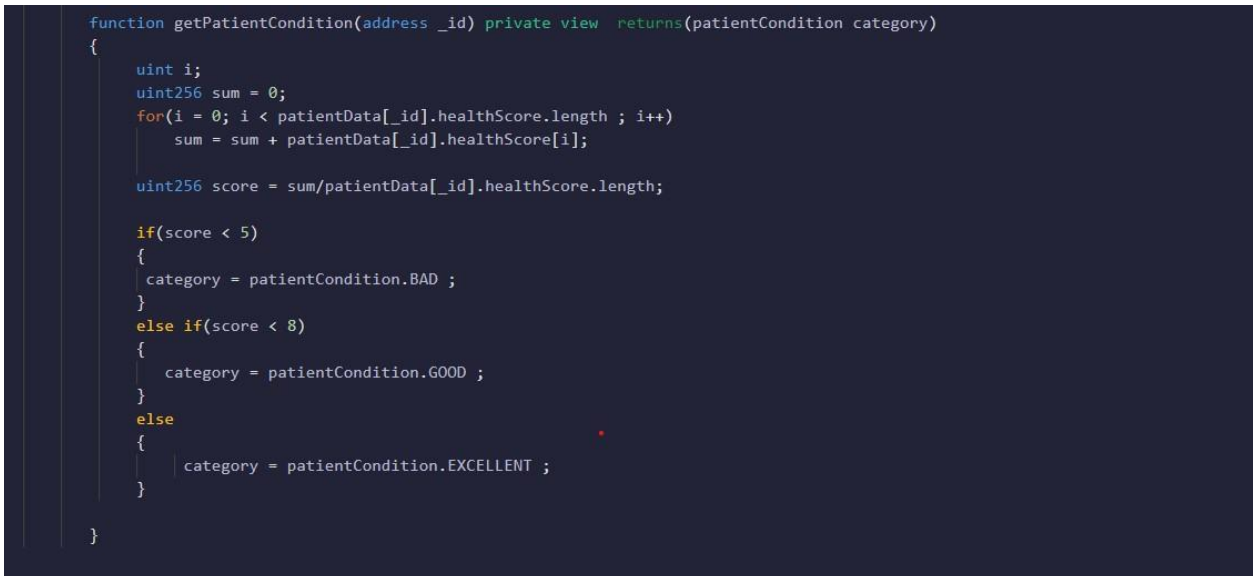


```
function enableAccess(address _id) public onlyPatients
{
    canSeePatient[msg.sender][_id] = true;
}
```

Enable Access can only be called by patients who want to give access to their data to some other stakeholders.

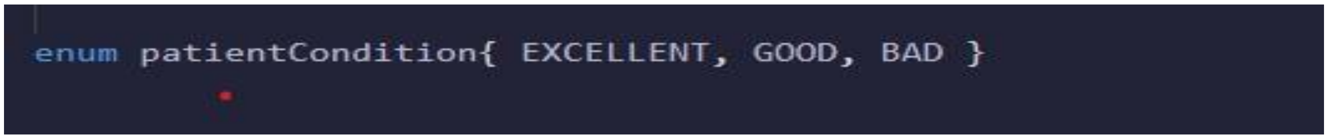
After the method has been invoked by people who have permission to see the patient data. This method will return the patient details like **health records**(as prescriptions),**name** of patient, and a boolean which is **isPatient**.

1. For Insurance Premium Calculation, this function calls another function which is **getPatientCondition** which returns the category of patient based on his previous health scores. **getPatientCondition** is a private function which can only be called from inside the contract. Screenshot is attached below:



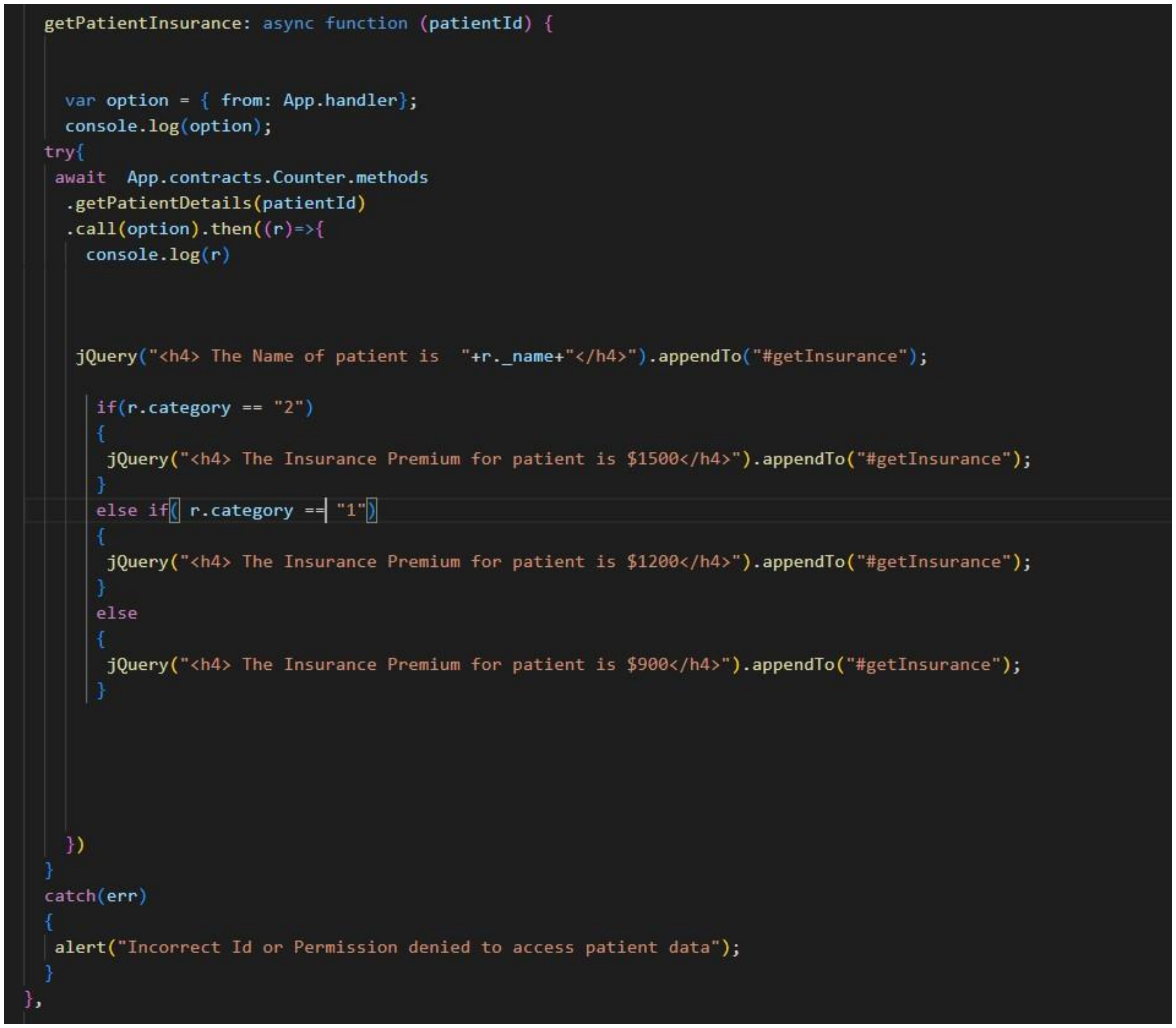
```
function getPatientCondition(address _id) private view  returns(patientCondition category)
{
    uint i;
    uint256 sum = 0;
    for(i = 0; i < patientData[_id].healthScore.length ; i++)
        sum = sum + patientData[_id].healthScore[i];

    uint256 score = sum/patientData[_id].healthScore.length;

    if(score < 5)
    {
     category = patientCondition.BAD ;
    }
    else if(score < 8)
    {
       category = patientCondition.GOOD ;
    }
    else
    {
         category = patientCondition.EXCELLENT ;
    }

}
```

This function returns **Category** which is an enum which can only have three values that is the patient condition can be Excellent, Good , Bad. Based on the patient condition which the smart contract evaluates based on previous health history we will display the insurance premium amount(the logic for premium decision is done on sever side & displayed on front end).

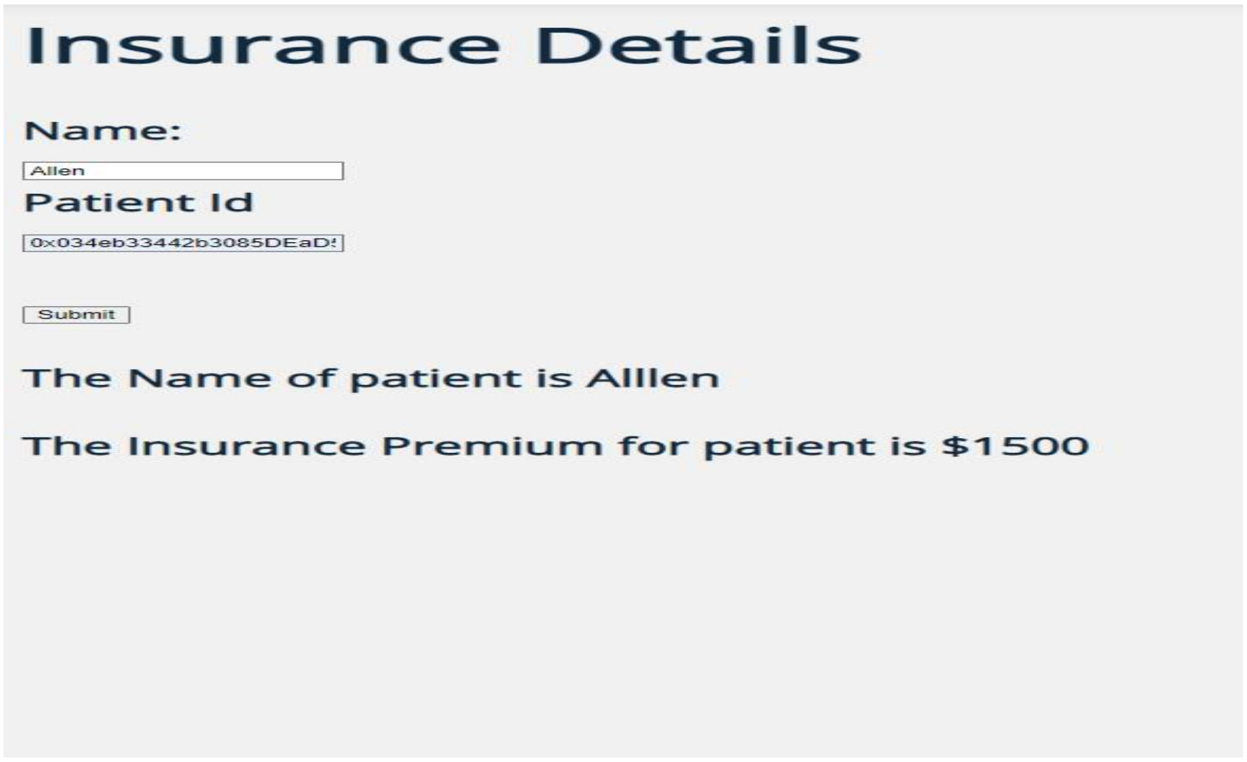


```
enum patientCondition{ EXCELLENT, GOOD, BAD }
```

The server side logic is shown below and how it is displayed on front end is shown below:



```
getPatientInsurance: async function (patientId) {


  var option = { from: App.handler};
  console.log(option);
try{
 await  App.contracts.Counter.methods
  .getPatientDetails(patientId)
  .call(option).then((r)=>{
    console.log(r)



   jQuery("<h4> The Name of patient is  "+r._name+"</h4>").appendTo("#getInsurance");

     if(r.category == "2")
     {
      jQuery("<h4> The Insurance Premium for patient is $1500</h4>").appendTo("#getInsurance");
     }
     else if( r.category == "1")
     {
      jQuery("<h4> The Insurance Premium for patient is $1200</h4>").appendTo("#getInsurance");
     }
     else
     {
      jQuery("<h4> The Insurance Premium for patient is $900</h4>").appendTo("#getInsurance");
     }





  })
}
catch(err)
{
 alert("Incorrect Id or Permission denied to access patient data");
}
},
```


Insurance Details

Name:

Allen

Patient Id

0x034eb33442b3085DEaD!

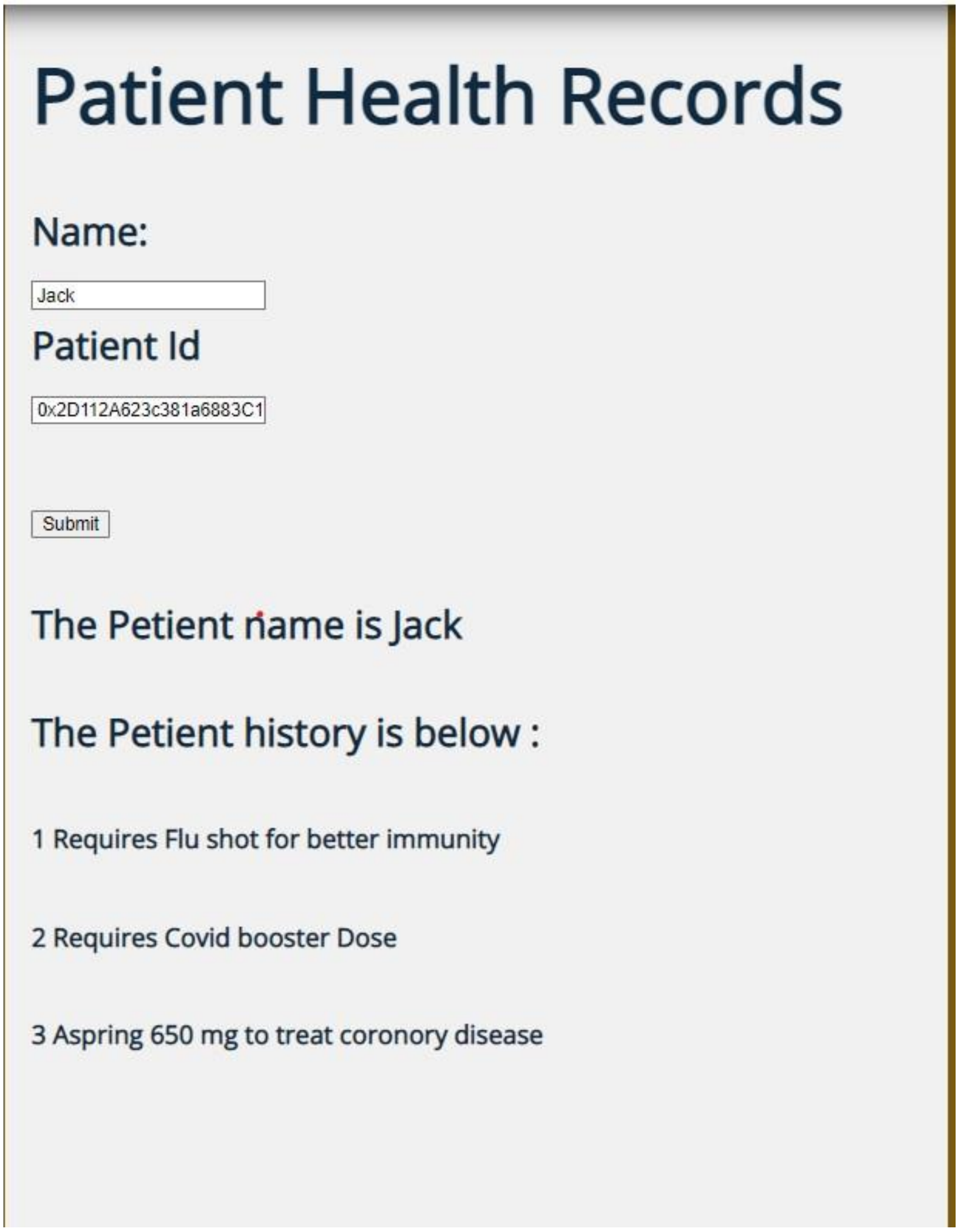
Submit

The Name of patient is Alllen

The Insurance Premium for patient is $1500

Right now we are just averaging the health scores & giving insurance premium based on certain ranges. But future plans Include to calculate insurance premium from health records by analyzing the health records as well as health score to make it more comprehensive.

1. For Patient History we retrieve similarly the health records & render the complete history on front end. Screenshot of patient history rendering is shown below:


Patient Health Records

Name:

Jack

Patient Id

0x2D112A623c381a6883C1

The Petient name is Jack

The Petient history is below :

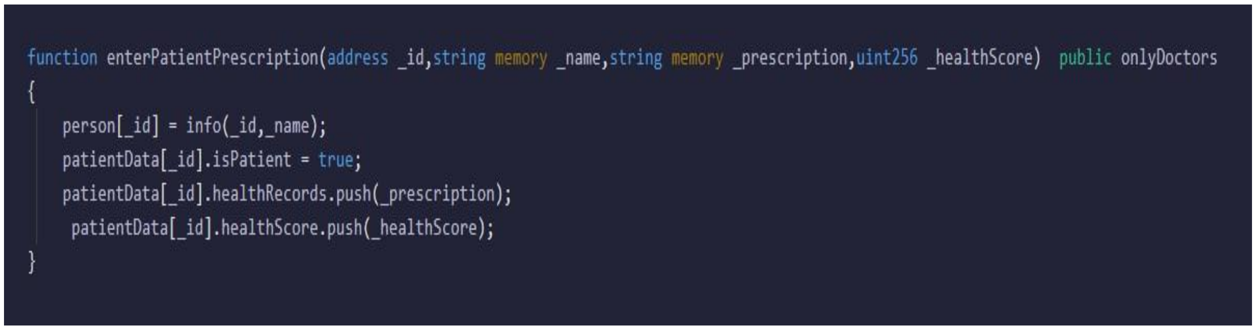
1 Requires Flu shot for better immunity

2 Requires Covid booster Dose

3 Aspring 650 mg to treat coronory disease

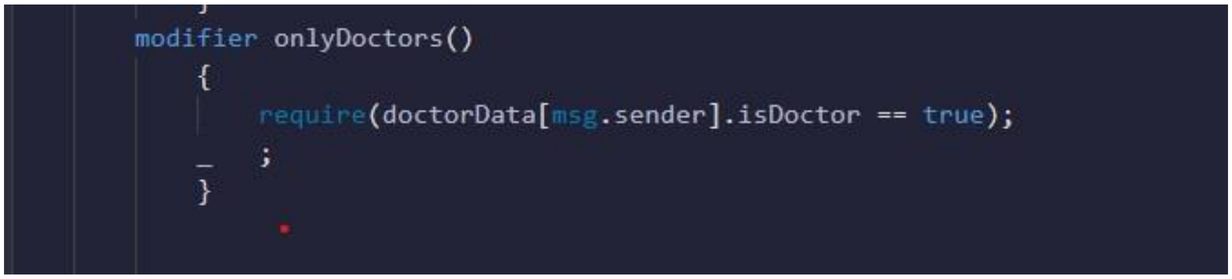
enterPatientPrescription:

One of the goals of our application is to achieve Patient Data Security & Data Integrity. This is the method which will ensure Data Integrity along with providing the base for other features of application like Data Privacy , Insurance Premium Calculation & Patient History Storage.



```
function enterPatientPrescription(address _id,string memory _name,string memory _prescription,uint256 _healthScore)  public onlyDoctors
{
    person[_id] = info(_id,_name);
    patientData[_id].isPatient = true;
    patientData[_id].healthRecords.push(_prescription);
     patientData[_id].healthScore.push(_healthScore);
}
```

The method can only called by doctors which are already registered in the Application. This is being made sure by calling the modifier onlyDoctors which checks whether the person who is calling this function is a Doctor. Since Data stored on blockchain will be immutable we are making sure only Doctors who are authenticated can add patient data on this application.



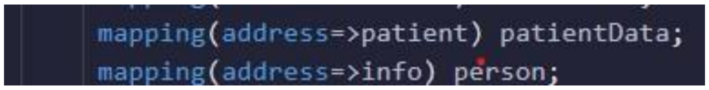
```
modifier onlyDoctors()
    {
        require(doctorData[msg.sender].isDoctor == true);
    _   ;
    }
```

This enterPatientPrescription method takes three parameters which are:

c. Address of the patient : This is the unique Identifier of the patient
d. Name of the patient :
e. Prescription for the Patient : Prescription for the patient
f. HealthScore of the Patient : The health score history will be used for insurance transparency & insurance calculation. The input here should be be integers from 1 to 10.

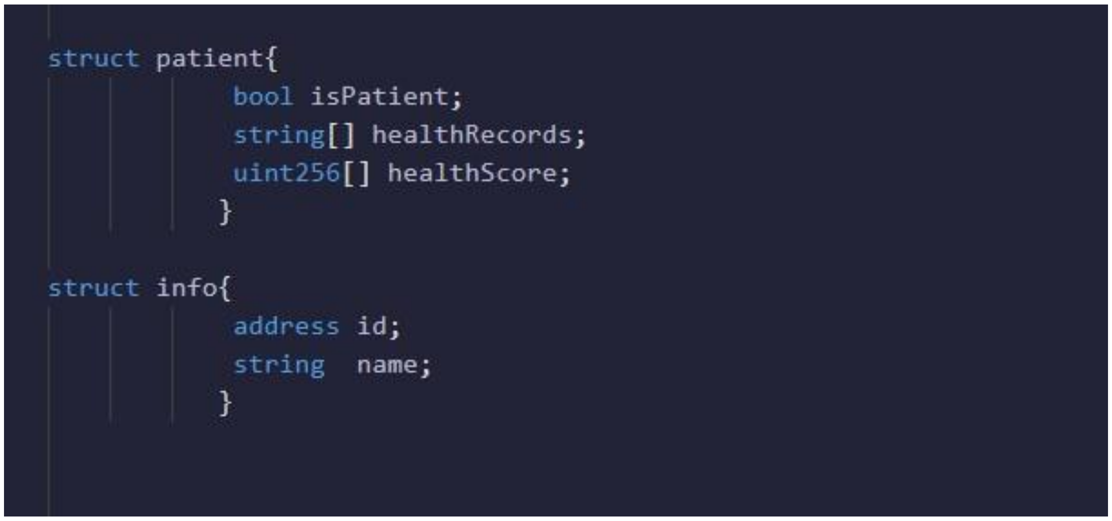
After the doctor calls this method the patient data is stored on the blockchain. The Method in turn stores this data into two mappings which map the address to a struct and are as below:

Mappings:



```
mapping(address=>patient) patientData;
mapping(address=>info) person;
```

Struct:

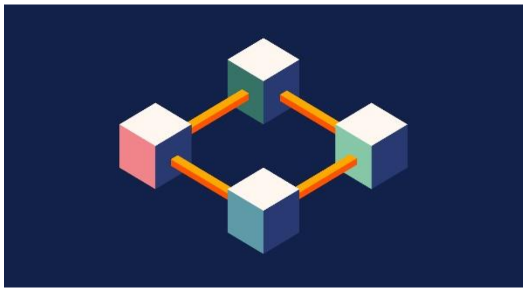


```
struct patient{
            bool isPatient;
            string[] healthRecords;
            uint256[] healthScore;
           }

struct info{
            address id;
            string  name;
           }
```

1. mapping(address=>info) person: This mapping will store the generic information of person. This mapping is also used while creating doctor.
   It does not store patient specific or doctor specific data. As we can see in above screenshot it just stores Name & ID, which can be for doctor as well as patient.
2. mapping(address=>patient) patientData:In this mapping patient specific data will be stored.
   a. First the isPatient variable will be made true as this is a patient.
   b. Second , in the healthRecords we will append the current prescription such that we have all the history of the patient.
   c. Third the health score also be previous healthscores & this will be later used to ensure Insurance Transparency to calculate insurance premium for patient.

10. <u>Token Design</u>

- The name we produced for now is Governmenterium and its symbol is



We named it Governmenterium because a blockchain-based digital government can protect data, streamline processes, and reduce fraud, waste, and abuse while simultaneously increasing trust and accountability. On a Governmenterium model, individuals, businesses, and governments share resources over a distributed ledger secured using cryptography. This structure eliminates a single point of failure and inherently protects sensitive citizen and government data.

The Governmenterium has the potential to solve legacy pain points and enable the following advantages:

- Secure storage of government, citizen, and business data
- Reduction of labor-intensive processes
- Reduction of excessive costs associated with managing accountability
- Reduced potential for corruption and abuse
- Increased trust in government and online civil systems

11. Data Structure Definitions

Data Structures:

1. String public name : This is a public variable that will store the name of the smart contract. In our smart contract the name is "HealthCareium"
2. address owner: This will store the address of the person who deploys the smart contract. In our case, the owner is the chairperson.
3. struct doctor: This will store multiple properties related to the person if the person is a doctor. It will have two properties described below
    3. bool isDoctor : This will be marked true for the doctor.
    3. string specialty: This property will store the specialty of the doctor
4. struct patient: This will store multiple properties related to the person if the person is a patient. It will have three properties described below

4. bool isPatient: Marked true for the patient
4. string[] healthRecords: This will store the health records of the patient
4. uint256[] healthScore: This will store the health score for the patient & it will be used to calculate insurance premium
5. struct info: This will store generic information of a person in the below properties. Here it will store common properties of the doctor as well as patient
5. address id: This will store the address of the patient
5. string name: This will store the name
6. enum patientCondition: This is our defined data type which will store the patient health condition category and the patient's health category will be decided based on his health scores. Also, these categories will define the insurance premium for patient
    6. EXCELLENT: Patient will get lowest insurance premium charge
    6. GOOD: Patient will get moderate insurance premium charge
    6. BAD: Patient will get the highest insurance premium charge
7. mapping(address=>doctor) doctorData: This will store data in key-value pairs. The key will be the address id of the patient and corresponding doctor data will be stored in value.
8. mapping(address=>patient) patientData: This will store data in key-value pairs. The key will be the address id of the doctor and corresponding patient data will be stored in value.
9. mapping(address=>info) person: This will store data in key-value pairs. The key will be the address id of the person, it can be a patient or doctor and corresponding data will be stored in value.
10. mapping (address => mapping (address => bool)) canSeePatient: This is double mapping where the first key will be the address of the patient. The inner key which is the second key will be the address of the person to whom the patient will provide permission to see his data. The value boolean will become true for the person to which the patient wants to give access.

MODIFIERS

1. modifier onlyOwner(): This modifier will ensure that the methods that are set to be executed by the owner, no other person can execute those. Creation of a doctor will use this modifier

2. modifier onlyPatients() :
This modifier will ensure that the methods that are set to be executed by the patients, no other person can execute those. Manage Acess will use this modifier.

3. modifier onlyDoctors():
This modifier will ensure that the methods that are set to be executed by the doctor, no other person can execute those. Creation of patient data will use this modifier.

Functions:

1. createDoctor : This function is used to create the doctor in our application. It will take input the doctor's name, address, and specialty of the doctor and store it in the doctor data mapping and person mapping corresponding to id.
2. getDoctorDetails: This function is used to get the doctor from our application. It will take input address and return whether the person is a doctor, the doctor's name, and his specialty.

3. enterPatientPrescription : This function can only be called by doctors and it will store the data of the patient in patient and person mappings. It will store data like name, address, append prescription, and health score.

4. getPatientDetails :This function is used to get patient details from our application. It will take input address and return whether the person is a patient, the patient's name and his health record, and the health condition category. This function can only be called by the patient or persons whom the patient has provided access.

5. enableAccess: This function can only be called by patients and it can manage who can see their data.

6. getPatientCondition: This is a private function that can only be called inside the smart contract. This will be called internally by the getPatientDetails function to decide which category patients' condition is based on the health score history. The categories are restricted through enum.

12. Solidity Code:

```
//SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;


        contract HealthCareium{
        string public name ;
        address owner;
        mapping(address=>doctor) doctorData;
        mapping(address=>patient) patientData;
        mapping(address=>info) person;
        mapping (address => mapping (address => bool))
canSeePatient;

        enum patientCondition{ EXCELLENT, GOOD, BAD }



                modifier onlyOwner()
                {
                require(msg.sender == owner);
                _    ;
                }
                modifier onlyPatients()
                {
                require(patientData[msg.sender].isPatient == true);
                _    ;
                }
                modifier onlyDoctors()
                {
                require(doctorData[msg.sender].isDoctor == true);
                _    ;
                }





                constructor() {
                owner = msg.sender;
                name = "HealthCareium";
                }

// All the Information regarding the person

                    struct doctor{
                    bool isDoctor;
                    string specialty;
                    }

```

```

struct patient{
bool isPatient;
string[] healthRecords;
uint256[] healthScore;
}
```

```

                    struct info{
                    address id;
                    string  name;
                    }




              function createDoctor(address _id,string memory
   _name,string memory _specialty)  public onlyOwner
              {
              person[_id] = info(_id,_name);
              doctorData[_id] = doctor(true,_specialty);
              }


              function getDoctorDetails(address _id)  public
              onlyOwner view returns(bool _isDoctor,string memory
              _name, string memory _specialty)
              {
              _isDoctor = doctorData[_id].isDoctor;
              _name = person[_id].name;
              _specialty = doctorData[_id].specialty;

           }


                function enterPatientPrescription(address _id,string
                memory _name,string memory _prescription,uint256
   _healthScore)  public onlyDoctors
                {
                person[_id] = info(_id,_name);
                patientData[_id].isPatient = true;
                patientData[_id].healthRecords.push(_prescription);
                patientData[_id].healthScore.push(_healthScore);
                }


                     function getPatientDetails(address
_id)  public  view  returns(bool _isPatient,string memory
_name,string[] memory _prescriptions,patientCondition category)
                    {
                    require((patientData[msg.sender].isPatient ==
                    true && _id == msg.sender) ||
                    canSeePatient[_id][msg.sender] == true,"You
                    dont have permission to view data");
                    _isPatient = patientData[_id].isPatient;
```

```
                        _name = person[_id].name;
                        _prescriptions =
                        patientData[_id].healthRecords;
                        category = getPatientCondition(_id);
                        }

                      function enableAccess(address _id) public
                      onlyPatients
                      {
                      canSeePatient[msg.sender][_id] = true;
                      }

                        function getPatientCondition(address _id)
                        private view  returns(patientCondition
                        category)
                        {
                        uint i;
                        uint256 sum = 0;
                        for(i = 0; i <
                        patientData[_id].healthScore.length ; i++)
                                                    sum = sum +
                        patientData[_id].healthScore[i];

                    uint256 score =
    sum/patientData[_id].healthScore.length;

    if(score < 5)
    {
    category = patientCondition.BAD ;
    }
    else if(score < 8)
    {
    category = patientCondition.GOOD ;
                                 }
else
    {
    category = patientCondition.EXCELLENT ;
    }
                 }
```

143. [redacted]

12. GVT(ERC 20) Code

Below we can see the complete GVT token code. Its an ERC 20 token and we have written the token contract in the same code as that of the HealthCareium Smart Contract because we wanted all the interaction to the token to happen through the HealthCareium Smart contract.
This would introduce Healthcareium Smart Contract as a hidden layer and HealthCareium contract will be the one who will deploy GVT Token so that it becomes the Owner of GVT. Because there are some functions that only owner can call to improve security & also we wanted user should not interact with wo contracts , he should just interact with one contract which is HealthCareium . So all the functions of the token that need to be executed have been indirectly used through HealthCareium smart contract.

```
//SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";


contract MyContract is ERC20{
    uint public number;
    address owner;

    modifier onlyOwner()
            {
                require(msg.sender == owner);
            _   ;
            }

    constructor() public ERC20("Governmenterium", "GVT") {
        _mint(msg.sender,10000000*10**18);
        owner = msg.sender;
    }

     // ERC20 Methods




           function rewardDoctor(address receiver,uint amount) public onlyOwner
           {

                 _mint(receiver,amount);
           }

            function rewardPatient(address receiver,uint amount) public onlyOwner
           {

                 _mint(receiver,amount);

           }

           function getInsurance(address sender,address agent,uint256 amount) public
onlyOwner
           {
              _transfer(sender,agent,amount);
           }

           function sellTokens(address receiver,uint256 amount) public onlyOwner
           {

                 _mint(receiver,amount);

           }


}


contract HealthCareium{


        MyContract public token;

         string public name ;
         address owner;
         mapping(address=>doctor) doctorData;
         mapping(address=>patient) patientData;
         mapping(address=>insuranceAgent) agentData;
         mapping(address=>info) person;
         mapping (address => mapping (address => bool))  canSeePatient;
         mapping(uint256 => address ) insurance;

        enum patientCondition{ EXCELLENT, GOOD, BAD }



        modifier onlyOwner()
            {
                require(msg.sender == owner);
            _   ;
            }
        modifier onlyPatients()
            {
                require(patientData[msg.sender].isPatient == true);
            _   ;
            }
        modifier onlyDoctors()
            {
                require(doctorData[msg.sender].isDoctor == true);
            _   ;
            }




           constructor() payable {
                owner = msg.sender;
                name = "HealthCareium";
                token = new MyContract();

            }

// All the Information regarding the person

        struct doctor{
                    bool isDoctor;
                    string specialty;
                   }


        struct patient{
                    bool isPatient;
                    string[] healthRecords;
                    uint256[] healthScore;
                   }

        struct insuranceAgent{
                    bool isInsuranceAgent;
                    string company;
                   }

        struct info{
                    address id;
                    string  name;
```

```
                  }



          function createDoctor(address _id,string memory _name,string memory
_specialty)  public onlyOwner
           {
               if(doctorData[_id].isDoctor == false)
                    {
                            token.rewardDoctor(_id,5000*10**18);
                    }
              person[_id] = info(_id,_name);
              doctorData[_id] = doctor(true,_specialty);
           }

            function createInsuranceAgent(address _id,string memory _name,string memory
_company)  public onlyOwner
           {

              person[_id] = info(_id,_name);
              agentData[_id] = insuranceAgent(true,_company);
              insurance[0] = _id;
           }


           function getDoctorDetails(address _id)  public onlyOwner view returns(bool
_isDoctor,string memory _name, string memory _specialty)
           {

              _isDoctor = doctorData[_id].isDoctor;
              _name = person[_id].name;
              _specialty = doctorData[_id].specialty;

           }

            function getAgentDetails()  public onlyOwner view returns(address adAgent)
           {
              adAgent = insurance[0];
           }


           function enterPatientPrescription(address _id,string memory _name,string memory
_prescription,uint256 _healthScore)  public onlyDoctors
           {
               if(patientData[_id].isPatient == false)
                    {
                            token.rewardPatient(_id,5000*10**18);
                    }
               person[_id] = info(_id,_name);
               patientData[_id].isPatient = true;
               patientData[_id].healthRecords.push(_prescription);
               patientData[_id].healthScore.push(_healthScore);
           }


           function getPatientDetails(address _id)  public  view  returns(bool
_isPatient,string memory _name,string[] memory _prescriptions,patientCondition category)
           {
               require((patientData[msg.sender].isPatient == true && _id == msg.sender) ||
canSeePatient[_id][msg.sender] == true,"You dont have permission to view data");
               _isPatient = patientData[_id].isPatient;
               _name = person[_id].name;
               _prescriptions = patientData[_id].healthRecords;
                category = getPatientCondition(_id);
           }

           function enableAccess(address _id) public onlyPatients
           {
               canSeePatient[msg.sender][_id] = true;
           }

           function getPatientCondition(address _id) private view  returns(patientCondition
category)
           {
                uint i;
                uint256 sum = 0;
                for(i = 0; i < patientData[_id].healthScore.length ; i++)
                    sum = sum + patientData[_id].healthScore[i];

                uint256 score = sum/patientData[_id].healthScore.length;

                if(score < 5)
                {
                 category = patientCondition.BAD ;
                }
                else if(score < 8)
                {
                   category = patientCondition.GOOD ;
                }
                else
                {
                     category = patientCondition.EXCELLENT ;
                }

           }
           // ERC20 Token Methods


            function getBalance(address _id) public view  returns(uint256 balance)
           {
               balance = token.balanceOf(_id);
           }

            function getInsurance(uint256 amount) public onlyPatients
           {
              token.getInsurance(msg.sender,insurance[0], amount*10**18);
           }

            function buyTokens(uint256 amount) public
           {
              token.sellTokens(msg.sender, amount*10**18);
           }









}
```

13. GVT Functions

Some of the basic function of the ERC 20 token that we use directly from the open zeppelin imported contract are balanceOf and _transfer. These are used using the IERC20 Interface provided for these functions.

1. balanceOf : This is used to get the balance of the user who is currenlt using the application . He can go to BUY GVT page and see his balance. Some users don't know how to use metamask that's why we gave this functionality at front end.

2. _transfer: This function is called to implement the payments part of the smart contract. As we have seen the moto of this app is insurance transparency and we are taking it to next level in this phase by introducing payments between the insurance company and user. So no third party charges or intermediatory charges are levied on them.

Some of the functions that we have defined in the contract are as follows:

1. rewardDoctor(): This is to implement reward part of application to attract as many doctors as possible. Whenever a doctor is registered he gets 5000 GVT. This function is not being directly called , here what happens is whenever we register a new doctor smart contract verifies it automatically and rewards the doctor by minting 5000 GVT for doctor.

```
  function createDoctor(address _id,string memory _name,string memory _specialty)  public
onlyOwner
           {
               if(doctorData[_id].isDoctor == false)
                    {
                            token.rewardDoctor(_id,5000*10**18);
                    }
              person[_id] = info(_id,_name);
              doctorData[_id] = doctor(true,_specialty);
           }
```

```
           function rewardDoctor(address receiver,uint amount) public onlyOwner
           {

                 _mint(receiver,amount);
           }
```

2. 1. rewardPatient(): This is to implement reward part of application to attract as many doctors as possible. Whenever a patient is registered he gets 5000 GVT. The rewarding mechanism happens in the same way as it happened for doctor.

3. getInsurance() : This is the part where payments come into play, this function will transfer the insurance amount in GVT from patient to insurance agent/company. Since our application is already having the health data and providing transparency by giving insurance premium rates based on health history. Using this function now the insurance premium calculated can be directly paid.

Again we are not directly calling the token getInsurance() function , here we first call the healthcareium getInsurance() function which inturns calls the token getInsurance Function.

Healthcareium getInsurance:

```
    function getInsurance(uint256 amount) public onlyPatients
           {
              token.getInsurance(msg.sender,insurance[0], amount*10**18);
           }
```

GVT getInsurance

```
function getInsurance(address sender,address agent,uint256 amount) public onlyOwner
           {
              _transfer(sender,agent,amount);
           }
```

4. sellTokens(): This function enables users to buy more GVT. We have enabled this in front end itself in buy GVT page. This function of the GVT is not directly called, the user can call buyTokens () of healthcareium which will inturn call this function

We have also implemented a modifier in smart contract which is onlyOwner() that will ensure we only the owner which is HealthCareium is able to call it.

14. <u>Deploying & Testing GVT ON REMIX</u>

For Deploying the token , follow the below steps:

a. Upload healthcareium.sol to remix. The file is present in the contracts folder.
b. Compile the contract.

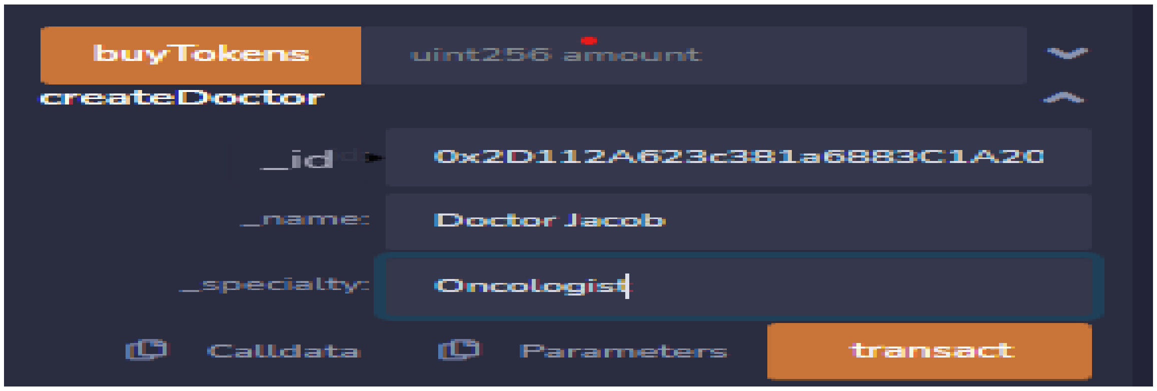
c. Deploy the contract. You have successfully deployed healthcareium smart contract as well as GVT smart contract. ALL GVT Token functions used for the SRAPP can be accessed from healthcareium smart contract functions.

For Testing the ERC 20 smart contract. It would be best if you have multiple(four) accounts imported to metamask , because the healthcareium smart contract as well as token contract needs to be tested from multiple personas. (Doctor , Owner , Patient , Insurance Agent). Otherwise for testing purpose you can deploy it on remix itself because remix provides with multiple dummy accounts.

Now below is the step by step guide for testing the application. The functions of token will be indirectly called by healthcareium smart contract. Please see previous two sections for more information.

Pre Requisite : The contract has been deployed using above mentioned steps on remix.

1. Create a doctor : Only owner(who deployed) the contract can create a doctor.



_id: 0x2D112A623c381a6883C1A20
_name: Doctor Jacob
_specialty: Oncologist

a. After creating a doctor we can check the balance of doctor by putting the address of doctor in getBalance function of healthcareium. As our SRAPP aims to attract consumers by rewarding them , as soon as doctor is registered he will get 5000 GVT. Here we have tested successfully the reward mechanism.
b. We are seeing 18 Zeroes infront of 5000 because that are the number of decimals we have defined for the token.

2. Create a Patient : Enter Patient details , If it’s a new patient he will automatically rewarded(validation done in smartcontract).

a. After creating a patient we can check the balance of patient by putting the address of patient in getBalance function of healthcareium. As our SRAPP aims to attract consumers by rewarding them , as soon as patient is registered he will get 5000 GVT. Here we have tested successfully the reward mechanism.

3. BuyTokens : Now this function of healthcareium will be used to buy tokens. Right now at smart contract level we are not doing any verification inside , the plan is to enable Dollar to GVT conversion in front end and verification will also be done in front end. So here we are simply calling buy function with amount .

a. If the transaction is successful , please check the balance & see whether the token has been credited.

4. Payments Testing :

a. First step is to create an insurance agent using admin. Insurance agents are not rewarded on our platform , so we can check the balance of insurance agent it should be 0.
b. Now we go back to the patient account.
c. Usually what will happen is on the front end we get the patient details that is the patient health category based on the patient health history stored on smart contract. And after this health condition , the insurance amount is calculated and when calling the getInsurance function the amount is calculated from end. But here we will manually check to see the transfer is taking place between insurance agent/company and patient.
d. We click on getInsurance() with amount and it will be transferred from patient to insurance agent. This can be verified by checking balance of both patient and insurance agent.

Using these 4 Steps , we have successfully verified Implementation of rewards and payments of GVT in the SRAPP.

15. Heroku Address

Below mentioned is the Heroku Endpoint . IF heroku starts charging then the end point might not be available . in that case , test on the local host.

https://cse-4-526-healthcareium.herokuapp.com/

16. References

- https://consensys.net/blockchain-use-cases/healthcare-and-the-life-sciences/
- https://youtu.be/CsxjlsBYmrI
- https://youtu.be/dvFOMm6mBao
- https://www.researchgate.net/figure/UML-Use-Case-for-Blockchain-BasedHealth-Application_fig2_361581795