ACKNOWLEDGEMENT

On the very outset of this report, we would like to extend our sincere and heartfelt obligation towards all the personages who have helped us in this endeavor. Without their active guidance, help, cooperation and engagement, we wouldn’t have made headway in the project.

We extend our gratitude in this project to its completion. Sincere apologies to those who weren’t mentioned to **Dr. Bisharathu Beevi** **A** principal, University College of Engineering, Kariavattom and **Mrs. Sangeetha S Nair** head of department, Information Technology for this opportunity.

We are ineffably indebted to our project coordinator **Mrs. Sabeena A S** and project guide **Prof. Arun B I** for their conscientious guidance and encouragement to accomplish this assignment.

We also remember with regard our parents, for all the help and moral support that they provided for implementing our project. At last but not least gratitude goes to all our friends who directly or indirectly helped us in this project to its completion.

Sincere apologies to those who weren’t mentioned.

CONTENTS

S. NO. TITLE PAGE NO.

**Abstract 4**

**List of figures 5**

**List of abbreviations 6**

**1. Introduction 7**

**2. Literature review 9**

**3. Methodology 10**

3.1 Algorithm 10

**4 Existing system 11**

4.1 Disadvantages of existing system 12

**5 Proposed System 13**

5.1 Proposed system’s architecture 15

5.2 Advantages of proposed system 16

**6. Design Environment 17**

6.1 Hardware Requirements 17

6.2 Software Requirements 17

6.3 Design 17

**7. Project description 21**

7.1 Modules 21

7.2 Process Involved 22

**8 Data Flow Diagram 23**

8.1 Definition 23

8.2 Purpose and Objective 23

8.3 DFD of the system 24

**9. Use Case 27**

**10 Software Design 28**

**11 Software Testing 3**2

11.1 Introduction 32

11.2 Objective 32

11.3 Test Plan 32

11.4 Unit Testing 33

11.5 Integration Testing 33

11.6 System Testing 33

**12 Limitations Of Proposed System 34**

**13 Conclusion & Future Enhancements 35**

13.1 Conclusion 35

13.2 Future Enhancements 35

**14 Appendix 36**

14.1 Source Code 36

14.2 Output Screenshots 109

**15. References 116**

ABSTRACT

The conventional system of patient-data management has several issues. The system includes information being stored as unstructured records presented as paper prescriptions, files and other traditional forms of storage. All the important data pertaining to the patient is stored by the centralized hospital authorities or the concerned medical practitioners. Reproducibility of this data when it comes to second – opinions or for the judgment of medical history, is a mammoth task. Even if there is a disease which is common, the treatment will mostly be not common for each individual as there should be considered the fact that there is a certain level of uniqueness with each patient. If a treatment strategy works on one patient, that does not mean it will work for all because there will be differences between each patient. Thus, the entire medical record history should be accessed so as to give the treatment which is best suited for the particular individual. Commonly, when a patient visits a new doctor, the doctor might recommend performing tests that have been previously performed. This might be because the proof of the previously conducted test cannot be produced as the test results might have been lost. This project deals with how Blockchain can be used to beat the odds faced by the conventional centralized system that greatly lacks interoperability. With the help of Blockchain, the patient’s data can be managed into a single record owned by the patient. The patient’s details pertaining to all healthcare services he/she has received will be managed into an easily accessible format for use anytime and anywhere. The project also deals with other aspects improving the interaction between the application and the patient, such as, real-world token tracking for appointments, appointment booking and so on. With the help of Blockchain, the current system can be completely disrupted and revolutionized, allowing for better transparency and ownership of the sensitive data thereby promoting and transforming the healthcare industries.

LIST OF FIGURES

**FIGURE NO. NAME PAGE NUMBER**

1. PROPOSED SYSTEM’S ARCHITECTURE 13

2. DFD LEVEL 1 19

3. DFD LEVEL 2 19

4. ADMIN PAGE 35

5. HOSPITAL MANAGEMENT PAGE 35

6. PATIENT PAGE 36

7. INSURANCE COMPANY PAGE 36

LIST OF ABBREVIATIONS

**dApp ………………………………………………. decentralized application**

**EVM ……………………………………………….Ethereum Virtual Machine**

**Web 3.0 …………………………………………………World Wide Web 3.0**

**POW ………….……………………………………………… Proof Of Work**

**POS ……………………………………………………………Proof Of Stake**

**P2P …..……………………………………………………………peer-to-peer**

**HTML ……………………………………… Hyper Text Mark-up Language**

**CSS …………………………………………………. Cascading Style Sheets**

**UI …………………………………………………………….. User Interface**

**DFD.................................................................................... Data Flow Diagram**

1. INTRODUCTION

The traditional approach to managing health records have been and inconvenient since its dawn. The amount of effort, time and space used up by traditional health information management systems are so massive that there is a great sense of wonder as to why a better system has not been introduced and implemented on a large scale. There are additional problems associated with the traditional paper-based management which include redundancy, proneness to loss of record, and so on. The use of technology in Blockchain is the health industry has the potential to have finest utility of Blockchain, since it involves store, use and transfer of sensitive information pertaining to any individual. Surprisingly, there has been little to no works or experiments done on this field as it comes up with many difficulties like scalability and awareness. A distributed platform providing technology like Blockchain, if utilized in health sector, and utilized properly, can yield amazing results in many aspects. The basic idea of our product is to minimize the effort and to overcome the aforementioned difficulties by making use of electronic health records to store and maintain the health information of every person. For the secure storage and transmission of sensitive health information, the blockchain technology, which is currently on an upsurge, is used.

2. LITERATURE REVIEW

Blockchain is a relatively new technology, which was conceptualized only in 2008 by a person or a group of people by the name of Satoshi Nakamoto. So, as the concept is relatively new, most of the individuals and even organizations focus on the one standout feature presented by Blockchain technology, which is crypto currency. Bitcoins and other forms of crypto currencies are still worth a lot and many individuals and organizations are still looking to invest in those crypto currencies and make tremendous profit from them. Other aspects to Blockchain technology are not yet fully discovered. This review dwells into the field of management of medical records which are electronic and which have the primary oversight on the efficiency of the system during emergency and catastrophic situations. A major part of the literature is based on software frameworks and other techniques introduced prior to blockchain and its capabilities of smart contracts. With the introduction of the ability to represent complicated data on the chain with the help of a language that is Turing-complete helped start a new field of distribution and p2p mode of communication. After the introduction of Ethereum , new software-frameworks that can use and employ blockchain have been developed by academic institutions and the IT-industry. Electronic Health Records and Electronic Medical Records are not the same thing . These terms are sometimes interchanged, but there is a big difference in the records containing medical information, stored digitally. An electronic medical record is the digital or electronic equivalent of the paper-records maintained by a patient and the doctor. It contains the history of the patient and other diagnostic and treatment details. The first system to use blockchain for Health records used a modular method for the sake of integration purposes. For the sake of scalability, the actual records are stored off-chain which is the provider’s RDB.

Blockchain contains the meta-data and other location information. In simple terms , a smart contract manages all the interaction between the participants of the system and defines the access matrix or access rules and other data-pointers. The pointers will contain tuples along with a query that will run on both the machines of the provider and the host. The health record software is designed according to the protocols of the network designed as Ethereum and the public as well as the private keys will have to decide which parties (network participants who act as miners of the system) get the permission. This means that every participant must have a node associated to the blockchain for interacting with the network. The concerning drawback of this kind of system is that every participant has to maintain a copy of the data. The other drawback would be the scalability issues because of the consensus mechanism used. If the host does not specify any limit, it is still possible to put a maximum transaction count per second of sixty. The projects were completed by focusing on data-sharing, accesscontrols and integration mechanism.

The research also focused on the patient-side, on how to ensure security constraints in the patient-data while aggregating the system. The various frameworks and blockchain software’s that have been developed so far can be categorized as two permissioned and the permissioned. In a permissioned network, since the participants know each there , it is possible to take advantage of the consensus mechanisms and any network interaction lag can be evaded from while also ensuring security, privacy and transparency. It is not associated to any cryptocurrency models, so the system does not need to be incentivized. This software-framework is most suitable for 2 or more organizations that know each other and want to transfer sensitive information.

3. METHODOLOGY

3.1 ALGORITHM

Consensus protocol is followed by every Blockchain network for making transactions secure and verified. A Consensus algorithm is a procedure through which all the peers of the Blockchain network reach a common agreement about the present state of the distributed ledger. Ethereum uses POS algorithm which was created as an alternative to POW.

**Steps:**

♣ The admin manages the system and adds hospital networks and insurance companies.

♣ The hospital network creates account for the patients and provides unique patient ID.

♣ The hospital network has the authority to edit and update patient data

♣ The patients are only allowed to view their data and possibly share it.

♣ The insurance companies verifies the patient data using patient ID and provide claims.

4. EXISTING SYSTEM

The conventional system of patient-data management has several issues. The system includes information being stored as unstructured records presented as paper prescriptions, files and other traditional forms of storage. All the important data pertaining to the patient is stored by the centralized hospital authorities or the concerned medical practitioners. Reproducibility of this data when it comes to second – opinions or for the judgment of medical history, is a mammoth task. Even if there is a disease which is common, the treatment will mostly be not common for each individual as there should be considered the fact that there is a certain level of uniqueness with each patient. If a treatment strategy works on one patient, that does not mean it will work for all because there will be differences between each patient. Thus, the entire medical record history should be accessed so as to give the treatment which is best suited for the particular individual. Commonly, when a patient visits a new doctor, the doctor might recommend performing tests that have been previously performed. This might be because the proof of the previously conducted test cannot be produced as the test results might have been lost.

4.1 DISADVANTAGES OF EXISTING SYSTEM

* Security concerns: Patient data is sensitive and confidential , so it’s critical to ensure that the system used to manage the data is secure. Unfortunately, many existing systems are vulnerable to data breaches, hacking and other security issues.
* Limited interoperability: Different healthcare providers often use different systems for patient data management, which can create challenges when trying to share data between them. This lack of interoperability can lead to fragmentation of patient records, which can impact the quality of care.
* Complexity: Many existing patient data management systems are complex
* Costs of manual medical records
* Implementation and Maintenance Requirements
* The cost of buying an EHR can be high, and the implementation process can slow down operations. You'll also need to consider how technical problems will impact your work. Will you be able to access records offline if your internet goes out or receive fast tech support when you need to troubleshoot? EHRs also require training and on going updates to keep the system prepared for new technological demands and policy changes, such as new requirements from HIPAA.
* Creating Unnecessary Patient Concerns: Because an electronic health record system enables patients to access their medical data, it can create a situation where they misinterpret a file entry. This can cause undue alarm, or even panic when the provider isn't there to clarify records

5.PROPOSED SYSTEM

Prior to the introduction of Blockchain and its Smart Contractcapabilities, the most widely discussed topic on Electronic Health Records was “How to store EMRs” whether to use a cloud based platform for storage or to use the localized systems itself. This meant centralization of information which indicated that every Health care Provider and hospital has to maintain all information pertaining to the patient records in their own premise that is the locally maintained storage and databases. The centralized model for storing patient record information has several issues associated with it, they are:

a). Not patient-driven: The patients do not have any control over the data as it is not owned by the patient. A patient’s data should be owned and controlled by the respective patient. As all records are made and stored in the hospital or healthcare service provider, the data is not technically owned by the patient. In order to improve privacy and security one should own one’s own healthcare data. A patient-centric model can disrupt the centralized manner of sensitive health care data storage.

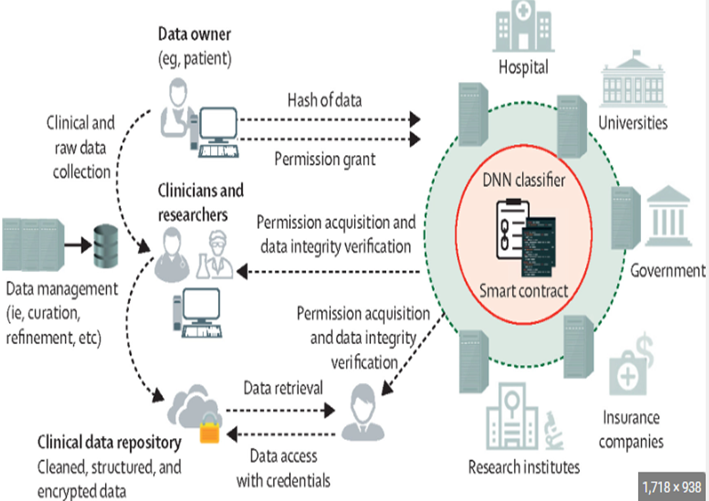
b). Scattering of records: The manner in which a patient receives treatment can varied and in different structures and this might cause the replicating of records.

c). Limitation in the interoperability of systems: All hospitals and healthcare service providers have different systems and methods for storing data . This will lead to issues in the sharing and viewing of data between the different healthcare service providers . A particular hospital’s system will only be equipped to view the details of that particular hospital. Patient transfers between hospitals may lead to redundant overheads.

d). Inconvenient secure sharing: The conventional ways and methods in which health care data is shared can be very complicated and time consuming . For example , Direct is an e-mail standard that allows physicians to transfer data via email in a secure manner. It encrypts the transmission between the physician and the receiver.

The solutions brought forward seemed to solve a lot of the early specified issues, but they suffered some form of vulnerability which led to the search of a better solution in the centralized lines leaving some or the other drawbacks unsolved, such as privacy, data-ownership and transparency. Furthermore, in scenarios like a disaster, the centralized model seemed to really suffer as the response is generally disorganized and any harm to the central storage can leverage a lot of important data. Even though natural calamities are events that are rare, the field of healthcare can greatly be leveraged by them by means of replication and sharing of the concerned information, the network grows powerful in the lines of reliability and robustness even in cases of huge failures. Also, peer to peer networks can allow ownership of data because sensitive data can be requested and stored only to the concerned system-node. A multiple number of such nodes can improve the accessibility of the stored information. Anyhow, this task of achieving consensus while maintaining privacy , security and anonymity can be extremely challenging. Blockchain technology has made it possible to achieve all these challenges in addition to improving transparency and reliability. Blockchain is a structure that stores data in a singly linked list manner as a sequence where every block is connected to the following block forming a chain . Breaching such a system will require rechaining all the blocks while maintaining consensus which is close to impossible.

5.1 PROPOSED SYSTEM ARCHITECTURE



5.2 ADVANTAGES OF PROPOSED SYSTEM

* Data is been added only by authorized bodies.
* Insurance company has the records of the hospital plus it can add data by user and verify if any record is tampered.
* Improves decision making as blockchain allows doctors from different location to view the same data in real time.
* Transforms the patient’s medical record to a decentralized system that can’t be tampered.
* Speeds up medical credentialing.
* Increases trust, security, transparency and the traceability of data shared across a business network.

6. DEVELOPMENT ENVIRONMENT

6.1 HARDWARE REQUIREMENTS

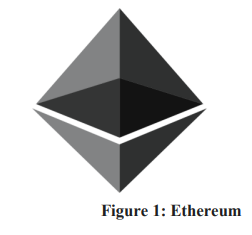
* Processor: Intel core i7( 6th gen or higher), 800MHz AMD
* 8 GB RAM
* 500 GB hard drive
* Stable internet connection

6.2 SOFTWARE REQUIREMENTS

* Ethereum
* Web3
* Ganache
* Truffle
* Smart Contract

6.3 DESIGN

1. Ethereum



Ethereum is the second largest platform for cryptocurrency, falling second to the bitcoin network. It is an open source and decentralized platform that provides the tools and requirements to build smart contracts. As reward for validating the transactions, the miners receive ether , which is the incentive of the system. Today, Ethereum is the platform for lakhs of cryptocurrencies, including three of the top leading cryptocurrencies. Ethereum network contains an Ethereum virtual machine that compiles scripts using a network of nodes connected internationally. The network also uses an internal transaction unit called gas that is used to allocate resources on the network. Ethereum platform was developed by Vitalik Buterin who is a cryptocurrency-researcher. In 2016 due to an exploitation in the smart contract of the DAO project and the theft of millions of dollars, Ethereum split into two different blockchains - ETH and ETC. The ETH had the theft reversed while the ETC continued along the same lines. Proof Of Work and the Proof Of Stake are the most widely used consensus algorithms in Ethereum. The POS consensus algorithm checks to see if the participant has high enough stakes of the concerned currency - this is a drawback as it opens the door to Monopoly, but POS has its own way to control it which is selecting a random stake holder on subsequent rounds.

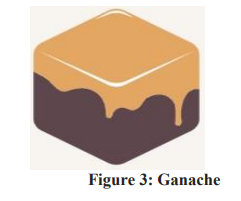
2. Web3



Figure 2: Web3

In order to communicate with the components in the chain, validations of the transactions should be done in chain. For a participant in the network of some other offline framework to create and validate a transaction, it has to relay it to the p2p network which is the underlying network. It also contains a library collection that facilitates the communication between the Ethereum nodes and the in-chain components .It is used in the server side for applications developed in Node.js. It connects to the Ethereum network with the help of an Ethereum-node using an HTTP connection. This can be a node in the local system provided by Infura or HD wallet. The integration of Ethereum and the web application can be done using Metamask which is an in- browser extension that allows to operate from Ethereum accounts. Metamask is an Ethereum wallet present in the browser that introduces the browser to a Web3 provider object. A Web3 provider provides a link to Ethereum nodes which are publicly accessible and is also a data- structure. With the help of metamask, a user can use, store and manage public and private keys which is unique to the account. The combination of Ethereum, metamask and web3js along with a web interface makes back end- front end communication very easy.

3. Ganache



Ganache is a local blockchain for the rapid development distributed applications on Ethereum. Ganache can be used throughout the development cycle so as to deploy, develop and test DAPPS in a deterministic and secure environment. Ganache User Interface is desktop app that supports both: Ethereum and Corda technology. Ethereum is also available as a command line version : Ganche-cli

4. Truffle



It is a powerful developing environment for Ethereum Virtual Machine (EVM) using blockchains and also acts as an asset pipeline and a test network/framework to the same. This component provides the following:

• The compiling , linking and development of smart contracts and the maintenance of binary dependencies.

• Automated smart contract test environment.

• Scriptable, extensible deployment & migrations framework.

• Management of Packages.

• Communication with Contracts directly.

• Build pipeline which are configurable with tight integration support. Truffle environment to run scripts.

5. Smart Contract

The smart contract used in Medicare makes the project patient centric . The patient is able to make all decisions pertaining to which medical practitioner can view and edit the record. On the patient’s end , there are 5 operations that correspond to the patient’s record : Per missioning and Revoking of View and Write Permissions to the Medical practitioner .The record details added by the medical practitioner will be stored in the blockchain .This acts as proof of existence for the recorded data. Only a validated practitioner can add recorded data and not any other network participant. The smart contract specifies the functionality for the access control. Access to a patient’s record can only be controlled by the patient.

7. PROJECT DESCRIPTION

7.1 MODULES

Admin Login

This is the admin prerogative of the patient data management system. Admin is required to login using username and password authentication. Matching username and password triggers response consequently logging into the system. An admin has access to sub modules:

* Hospital Network Management: To add, delete or update prospective hospital details.
* Insurance Provider Management: To add, delete or update insurance providers details.

Hospital Management Sign-Up

This module helps the hospital management to login to the system . The hospital management helps to create account for patients within our system. The patients are provided with a unique patient ID through which other activities related to patient data management could be accessed.

Insurance Company Sign-Up

This module helps the registered and approved insurance companies to login using their username and password. After logging in they can view the individual patient records using the unique patient ID.

7.2 PROCESSES INVOLVED

* **Decentralized Electronic Health Records (EHRs):** Blockchain can serve as a tamper-proof ledger for storing patient health records. Each patient may have their own encrypted EHR, and access to this data can be controlled through cryptographic keys.
* **Identity and Access Management:** Blockchain can be used to establish a unique patient identity that is cryptographically secure. Access management modules allow patients to control who can view and modify their health data.
* **Consent Management:** Smart contracts on the blockchain can facilitate patient consent management. Patients can define the terms under which their data can be accessed or shared.
* **Interoperability:** Blockchain can act as a bridge between different healthcare systems, making it easier to exchange data between hospitals, clinics, and other healthcare providers.
* **Data Integrity and Immutability:** Blockchain's decentralized nature ensures that data cannot be altered or deleted once it is recorded, providing a reliable audit trail for patient data.
* **Secure Data Sharing:** Blockchain technology can enable secure and auditable data sharing between authorized parties while maintaining patient privacy.
* **Medical Research and Clinical Trials:** Blockchain can facilitate the collection of patient data for medical research and streamline the process of conducting clinical trials by securely managing patient consent and data sharing.
* **Healthcare Payments and Billing**: Blockchain can be used for transparent and secure healthcare billing and payments, reducing fraud and administrative costs.
* **Audit and Compliance**: Blockchain provides a transparent and tamper-resistant record of data access and modifications, helping healthcare organizations meet regulatory compliance requirements.

8. DATA FLOW DIAGRAM [DFD]

8.1 DEFINITION

A Data Flow Diagram (DFD) is a way of representing a flow of data through a process or an information system. The DFD also provides information about the outputs and inputs of each entity and the process itself. A DFD has no control flow – there are no decision rules and no loops. Specific operations based on the data can be represented by a flowchart. For each data flow, at least one of the endpoints (source/destination) must exist in a process. The refined representation of a process can be done in another data-flow diagram, which subdivides this process into sub-processes. To make DFD more transparent (i.e., not too many processes), multi-level DFDs are at a higher level are less detailed (aggregate more detailed DFD at lower levels). The contextual DFD is the highest in the hierarchy. The so-called zero level is followed by DFD 0. Next, first level-DFD 1 and so on. The number of levels depends on the size of the model system. Zeroth level DFD contains the most important system functions. The lowest level should include processes that make it possible to create a process specification.

8.2 PURPOSE AND OBJECTIVE

♣ Main objective is to represent the processes and data flow between them

♣ It explains the flow and process of data input, data output, and storing data

♣ Symbols used in DFD are: rectangles (represent the data entity), circles (represent the process), arrows (represent the flow of data), ovals or parallel lines (represent data storing)

♣ Rule followed by DFD is that at least one data flow should be there entering into and leaving the process or store

♣ It models the flow of data through a system

8.3 DFDs OF THE SYSTEM

Patient Flowchart

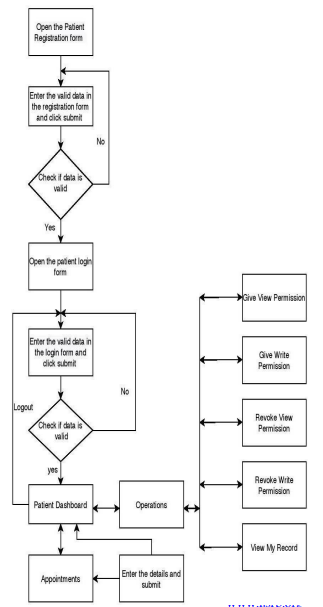
The patient’s has mainly four operations to perform are:

a). Give View Permission: Permissions a Doctor/Practitioner to view the record. This function takes the doctor’s Address as input which is shared to the patient offline. The address is unique to the practitioner which maps to the corresponding Ethereum account.

b). Give Write Permission: Permissions a Doctor/Practitioner to Write to the record. This function takes the doctor’s Address as input which is shared to the patient offline. The address is unique to the practitioner which maps to the corresponding Ethereum account. Only a permissioned practitioner can write to the record.

c). Revoke View Permission: Revokes View permission from a practitioner who was earlier given permission to view the record, by the patient. Permission can only be revoked from a practitioner who had the View Permission already. The practitioner will no longer be a participant in the patient’s private chain unless permissioned by the patient.

d). Revoke Write Permission: Revokes Write permission from a practitioner who was earlier given permission to write to the record, by the patient. Permission can only be revoked from a practitioner who had the Write Permission already. This function will deny the doctor/practitioner from adding any further record details to the patient’s private chain. The practitioner will no longer be a participant in the patient’s private chain unless permissioned by the patient.

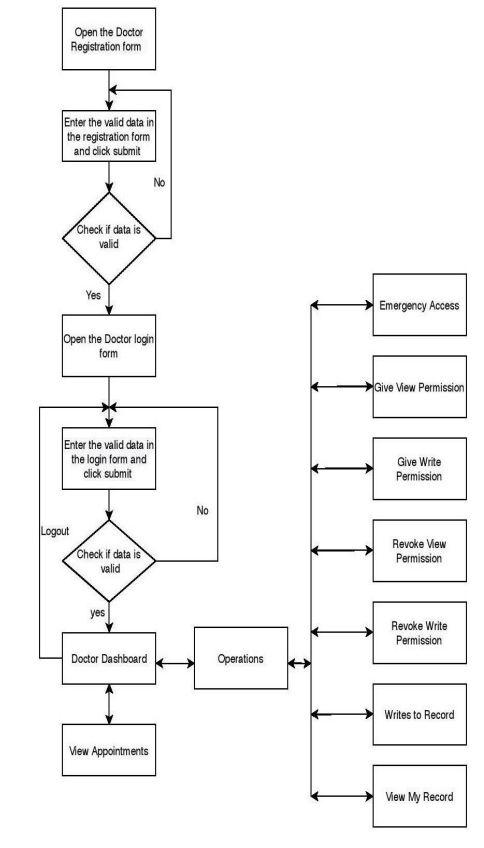


Practitioner’s Flowchart

The practitioner can also be a patient to another doctor/practitioner. Along with all the functions included in the patient-end , the practitioner has two additional functions are:

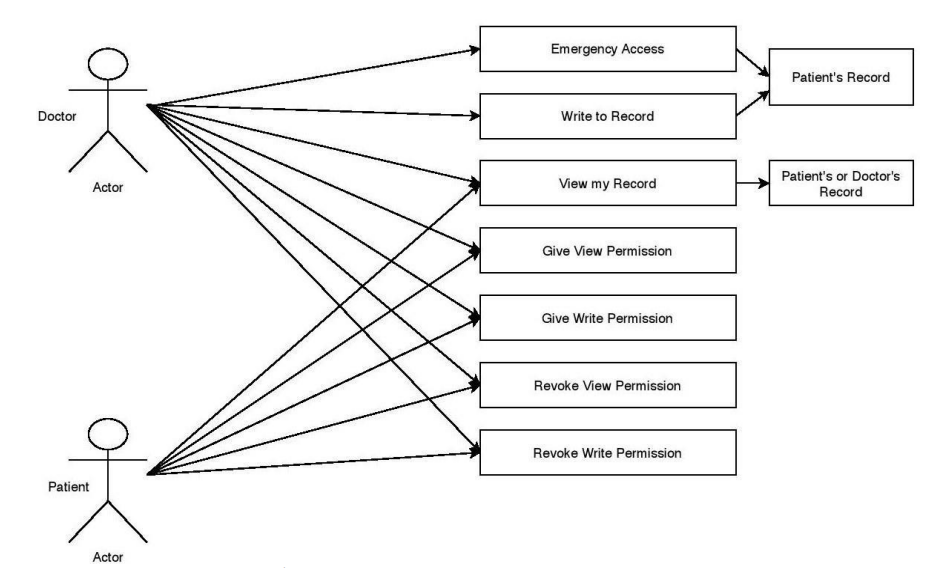
a) Write to Record : The practitioner can write to the patient’s record. All this information will be stored in the private chain of the patient. A doctor can , at a time, be permissioned to multiple patient-accounts depending on the number of patient the doctor can tend to.

b) Emergency Access: In case of an emergency situation where the patient is incapacitated or for any other reason the patient is unable to provide view permission to the doctor , the doctor can invoke emergency access to view lifesaving and sensitive information about the permission that will help with the treatment of the patient. When such an access has been invoked , an alert will be sent to the patient’s account which can be viewed by the patient whenever the patient is able to do so . The patient can then further authorize the permission or revoke it. If the access was illegitimate, the doctor’s permission will be revoked by the patient and this can be used as proof for any law proceedings.



9. USE CASE

The use-case of this application has two main entities as actors, one is the patient and the other is the doctor. Now, it is to be noted that emergency access, which is a special feature given to the doctor is also included in the list of operations. Out of the seven of the list of operations, five can be accessed by the patient while the doctor has access to all of the seven operations. The operation which can bring a change in the block data once accessed is write to patient's record operation which can be done only by the doctor. Giving and revoking view or write permissions can affect the network in a way that changes the corresponding doctor's view of access to the corresponding operations. The Figure 7 represents the Use case diagram.



10.SOFTWARE DESCRIPTION

The following software’s are used in the development of this project:

**1. VISUAL STUDIO CODE**

Visual Studio Code is a source-code editor that can be used with a variety of programming languages, including Java, JavaScript, Go, Node.js, Python, C++ and Fortran. It is based on the Electron framework which is used to develop Node.js Web applications that run on the Blink layout engine. Visual Studio Code employs the same editor component (codenamed "Monaco") used in Azure DevOps (formerly called Visual Studio Online and Visual Studio Team Services).

Out of the box, Visual Studio Code includes basic support for most common programming languages. This basic support includes syntax highlighting, bracket matching, code folding, and configurable snippets. Visual Studio Code also ships with IntelliSense for JavaScript, TypeScript, JSON, CSS, and HTML, as well as debugging support for Node.js. Support for additional languages can be provided by freely available extensions on the VS Code Marketplace.

Visual Studio Code can be extended via extensions, available through a central repository. This includes additions to the editor and language support. A notable feature is the ability to create extensions that add support for new languages, themes, debuggers, time travel debuggers, perform static code analysis, and add code linters using the Language Server Protocol

**2. Node.js**

Node.js is an open-source, server-side runtime environment that allows developers to execute JavaScript code outside of a web browser. It is built on the V8 JavaScript engine, which is the same engine that powers Google Chrome, and it enables developers to run JavaScript on the server-side, rather than just on the client-side in the browser.

Node.js was created by Ryan Dahl and was first released in 2009. It quickly gained popularity among developers due to its performance, scalability, and asynchronous event-driven architecture. The key features of Node.js include:

1. Asynchronous I/O: Node.js is designed to handle asynchronous operations efficiently, allowing it to handle a large number of concurrent connections without blocking the execution of code.
2. Non-blocking event loop: Node.js operates on an event-driven, non-blocking architecture, which means that it processes multiple requests concurrently without waiting for one operation to complete before moving on to the next one. This design is particularly suitable for handling real-time applications and APIs.
3. NPM (Node Package Manager): Node.js comes with a package manager called NPM, which is one of the largest software registries. NPM allows developers to access and use a wide range of open-source libraries and tools, simplifying the development process.
4. Cross-platform: Node.js is cross-platform and can run on various operating systems, such as Windows, mac OS, and Linux.
5. Web server capabilities: Node.js can be used to create web servers to handle HTTP requests, making it an excellent choice for building web applications and APIs.
6. Command-line tool: Node.js can also be used as a command-line tool to execute JavaScript code outside of a browser environment.

Node.js has found extensive use in building web servers, real-time applications (e.g., chat applications, gaming servers), command-line tools, and backend services for web applications. It has become a popular choice for developers looking to use a single language (JavaScript) for both frontend and backend development, facilitating code reuse and sharing expertise across the development stack

**3 MongoDB**

MongoDB is a popular, open-source, NoSQL (non-relational) database management system that provides a flexible and scalable approach to storing and managing data. It was developed by MongoDB Inc. and was first released in 2009.

Key features of MongoDB include:

1. Document-Oriented: MongoDB is a document database, which means it stores data in JSON-like documents. These documents can have varying structures, making it highly flexible and suitable for storing complex and hierarchical data.
2. NoSQL: MongoDB is part of the NoSQL database family, which diverges from traditional relational databases. NoSQL databases like MongoDB are designed to handle large amounts of unstructured or semi-structured data and can scale horizontally to accommodate growing datasets.
3. Scalability: MongoDB is built to scale horizontally by sharding data across multiple servers. This allows it to handle large amounts of data and high traffic loads efficiently.
4. Replication: MongoDB supports automatic data replication, providing data redundancy and high availability. This feature ensures that if one server fails, data can be seamlessly retrieved from other nodes in the replica set.
5. Ad hoc Queries: MongoDB supports dynamic queries on documents using a rich set of query operators, enabling developers to retrieve specific data without the need for complex JOIN operations often required in relational databases.
6. Indexing: MongoDB supports various types of indexes, such as single-field, compound, geospatial, and text indexes, which can significantly improve query performance.
7. Aggregation Framework: MongoDB provides a powerful aggregation framework that allows users to perform advanced data aggregation and transformation operations within the database.
8. GridFS: MongoDB's GridFS is a file storage system that allows you to store and manage large files, such as images, videos, and audio files, directly within the database.

MongoDB is widely used in various applications and industries, including web applications, mobile apps, content management systems, IoT (Internet of Things) platforms, real-time analytics, and more. Its flexibility, scalability, and ease of use have made it a popular choice for developers looking for a NoSQL solution to handle modern data requirements.

11. SYSTEM TESTING

11.1 INTRODUCTION

System Testing is a level of testing that validates the complete and fully integrated software product. The purpose of a system test is to evaluate the end-to-end system specifications. Usually, the software is only one element of a larger computer-based system. Ultimately, the software is interfaced with other software/hardware systems. System Testing is actually a series of different tests whose sole purpose is to exercise the full computer-based system. Black box testing and White box testing are two categories of testing. System test falls under the black box testing category of software testing. White box testing is the testing of the internal workings or code of a software application.

11.2 OBJECTIVE

♣ Reducing risks, for bug-free components don’t always perform well as a system.

♣ Preventing as many defects and critical bugs as possible by careful examination.

♣ Verifying the conformance of design, features, and performance with the specifications stated in the product requirements.

♣ Validating the confidence in the system as a whole before moving to the final stage – acceptance testing that takes place right before users get access to a product.

11.3 TEST PLAN

A Test Plan was created prior to the development stage that describes the test strategy, objectives, schedule, estimation, deliverables, and resources required to perform testing for a software product. Test Plan helped us to determine the effort needed to validate the quality of the application under test. The test plan served as a blueprint to conduct software testing activities as a defined process, which is minutely monitored and controlled by the test manager.

11.4 UNIT TESTING

Unit testing was conducted to validate that each unit of the software code performs as expected. Unit Testing is done during the development (coding phase) of an application by the developers. Unit Tests isolated a section of code and verify its correctness. A unit might be an individual function, method, procedure, module, or object.

11.5 INTEGRATION TESTING

Software modules were integrated logically and tested as a group. A typical software project consists of multiple software modules, coded by different programmers. The purpose of this level of testing is to expose defects in the interaction between these software modules when they are integrated.

11.6 SYSTEM TESTING

We validated the complete and fully integrated software product. The purpose of a system test is to evaluate the end-to-end system specifications. Usually, the software is only one element of a larger computer-based system. Ultimately, the software is interfaced with other software/hardware systems. System Testing is actually a series of different tests whose sole purpose is to exercise the full computer-based system.

12. LIMITATIONS OF THE PROPOSED SYSTEM

♣ **Maintenance** - dapps are hard to maintain because the code and data published to the blockchain are harder to modify. It’s not easy for developers to make updates to their dapps (or the underlying data stored by a dapp) once they are deployed - even if bugs or security risks are identified in an old version.

♣ **Performance overhead** - There is a huge performance overhead, and scaling is really difficult. To achieve the level of security, integrity, transparency, and reliability that Ethereum aspires to, every node processes and stores every transaction. On top of that, proof-of-work takes a long time. A back-of-the-envelope measurement puts the overhead at something like 1,000,000x that of standard computation currently.

♣ **Network congestion** - When one dapp uses many computational resources, the entire network gets backed up. Currently, the network can only process around 10-15 transactions per second; if transactions are being sent faster than this, the pool of unconfirmed transactions can quickly balloon.

♣ **User experience** - It can be harder to engineer user-friendly experiences because the average end-user might find it too hard to set up a tool stack necessary for interacting with the blockchain in a truly secure fashion.

13. CONCLUSION AND FUTURE ENHANCEMENT’S

13.1 COCLUSION

The traditional system for maintaining the records in medical sector is difficult and it requires large space to store the results of medical test for all patients. In previously used systems the data is in unstructured manner and it is difficult to exchange the data. So, for solving the above issues we plan on implementing the given model for managing health records in block chain usingMedicare. The EHR using blockchain is a revolution in the medical industry. It solves most challenges which exist today just in the name of trust in the medical sector. It not only provides a reliable platform for patient data exchange, but also a faster and empowering system. The time and effort expended in managing patient data, can be greatly minimized using EHR with effective and efficient results.

13.2 FUTURE ENHANCEMENT’S

* Insurance Companies could use this system for granting claims to their cutomer’s by checking their medical records.
* A provision for laboratories could also be included in this system so that they could update the report’s of patient directly to the system.
* All the activities which requires body check up’s colud utilise this system.

14. APPENDIX

14.1 SOURCE CODE

* **Related functions of Admin , Hospital network , Patient and Insurance providers**

**1 ADMIN**

const Admin = require("../models/adminModel");

const Hospital = require("../models/hospitalModel");

const Insurance = require("../models/insuranceModel");

const { v4: uuidv4 } = require("uuid");

const {

createHospitalService,

fetchAllHospitalsService,

createInsuranceCompanyService,

} = require("../services/blockchain/blockchainService");

const UsedAddress = require("../models/usedAddressModel");

const adminSignInController = async (req, res) => {

const { email, password } = req.body;

const admin = await Admin.findOne({ email });

if (admin) {

if (password === admin.password) {

res.status(200).json({

status: "success",

message: "Login Sucess",

role: "admin",

id: admin.adminId,

});

} else {

res.status(400).json({ status: "failed", message: "Invalid Password" });

}

} else {

res.status(400).json({ status: "failed", message: "Invalid Email" });

}

};

const adminSignOutController = async (req, res) => {};

const adminCreateHospitalController = async (req, res) => {

const {

name,

email,

password,

address,

state,

phone,

wallet: hospitalWalletAddress,

adminId,

} = req.body;

const admin = await Admin.findOne({ adminId });

const hospital = await Hospital.findOne({ email });

if (

!name ||

!email ||

!password ||

!address ||

!state ||

!phone ||

!hospitalWalletAddress ||

!adminId

) {

const response = {

status: "failed",

message: "insufficient information",

};

res.status(400).json({

response,

});

} else if (!admin) {

const response = {

status: "failed",

message: "admin with the id does not exist",

};

res.status(409).json({

response,

});

} else if (hospital) {

const response = {

status: "failed",

message: "hospital email already in use",

};

res.status(409).json({

response,

});

} else {

const hospitalId = uuidv4();

const adminWalletAddress = admin.wallet;

createHospitalService(adminWalletAddress, hospitalWalletAddress, hospitalId)

.then(async (response) => {

if (response.status != "success") {

console.log(response);

res.status(404).json(response);

}

const hospital = new Hospital({

hospitalId,

name,

email,

password,

address,

state,

phone,

wallet: hospitalWalletAddress,

});

const usedAddress = new UsedAddress({ address: hospitalWalletAddress });

await usedAddress.save();

const result = await hospital.save();

const { transactionHash } = response.data;

const { password: rmPass, ...hospitalData } = result.\_doc;

const responseObj = {

status: "success",

message: "created hospital successfully",

data: {

transactionHash,

...hospitalData,

},

};

res.status(200).json(responseObj);

})

.catch((err) => {

console.log(err);

const response = {

status: "failed",

message: "Internal error",

data: {

error: err.message,

},

};

res.status(500).json({

response,

});

});

}

};

const adminDeleteHospitalController = async (req, res) => {

try {

const deletedHospital = await Hospital.findByIdAndRemove(req.body.id);

if (!deletedHospital) {

return res.status(404).json({ error: "Hospital not found" });

}

res.json({ message: "Hospital deleted successfully" });

} catch (err) {

console.log(err);

res.status(500).json({ error: "Failed to delete hospital" });

}

};

const adminCreateInsuranceController = async (req, res) => {

const {

name,

email,

password,

address,

state,

phone,

wallet: insuranceWalletAddress,

adminId,

} = req.body;

const admin = await Admin.findOne({ adminId });

const insurance = await Insurance.findOne({ email });

if (

!name ||

!email ||

!password ||

!address ||

!state ||

!phone ||

!insuranceWalletAddress ||

!adminId

) {

const response = {

status: "failed",

message: "insufficient information",

};

res.status(400).json({

response,

});

} else if (!admin) {

const response = {

status: "failed",

message: "admin with the id does not exist",

};

res.status(409).json({

response,

});

} else if (insurance) {

const response = {

status: "failed",

message: "insurance company email already in use",

};

res.status(409).json({

response,

});

} else {

const insuranceCompanyId = uuidv4();

const adminWalletAddress = admin.wallet;

createInsuranceCompanyService(

adminWalletAddress,

insuranceWalletAddress,

insuranceCompanyId

)

.then(async (response) => {

if (response.status != "success") {

console.log(response);

res.status(404).json(response);

}

const insuranceCompany = new Insurance({

insuranceCompanyId,

name,

email,

password,

address,

state,

phone,

wallet: insuranceWalletAddress,

});

const usedAddress = new UsedAddress({

address: insuranceWalletAddress,

});

await usedAddress.save();

const result = await insuranceCompany.save();

const { transactionHash } = response.data;

const { password: rmPass, ...insuranceCompanyData } = result.\_doc;

const responseObj = {

status: "success",

message: "created insurance company successfully",

data: {

transactionHash,

...insuranceCompanyData,

},

};

res.status(200).json(responseObj);

})

.catch((err) => {

console.log(err);

const response = {

status: "failed",

message: "Internal error",

data: {

error: err.message,

},

};

res.status(500).json({

response,

});

});

}

};

const adminDeleteInsuranceController = async (req, res) => {

try {

const deletedInsurance = await Insurance.findByIdAndRemove(req.body.id);

if (!deletedInsurance) {

return res.status(404).json({ error: "Insurance not found" });

}

res.json({ message: "Insurance deleted successfully" });

} catch (err) {

console.log(err);

res.status(500).json({ error: "Failed to delete Insurance" });

}

};

const adminFetchAllHospitalController = async (req, res) => {

try {

const allHospitalsData = await Hospital.find().lean();

res.json({

status: "success",

message: "Hospitals Data Fetched sucessfully",

data: allHospitalsData,

});

} catch (err) {

console.log(err);

res

.status(500)

.json({ status: "success", message: "Internal Server Error" });

}

};

const adminFetchAllInsuranceController = async (req, res) => {

try {

const allInsuranceData = await Insurance.find().lean();

res.json({

status: "success",

message: "Insurance Data Fetched sucessfully",

data: allInsuranceData,

});

} catch (err) {

console.log(err);

res

.status(500)

.json({ status: "success", message: "Internal Server Error" });

}

};

module.exports = {

adminSignInController,

adminSignOutController,

adminCreateHospitalController,

adminDeleteHospitalController,

adminCreateInsuranceController,

adminDeleteInsuranceController,

adminFetchAllHospitalController,

adminFetchAllInsuranceController,

};

**2 Hospital Management**

const Patient = require("../models/patientModel");

const Hospital = require("../models/hospitalModel");

const { v4: uuidv4 } = require("uuid");

const {

createPatientService,

updatePatientReportService,

addAuthorizedHospitalService,

addAuthorizedInsuranceCompanyService,

fetchAuthorizedHospitalsService,

fetchAuthorizedInsuranceCompaniesService,

fetchAllPatientsService,

} = require("../services/blockchain/blockchainService");

const Insurance = require("../models/insuranceModel");

const UsedAddress = require("../models/usedAddressModel");

const hospitalSignInController = async (req, res) => {

const { email, password } = req.body;

const hospital = await Hospital.findOne({ email });

if (hospital) {

if (password === hospital.password) {

res.status(200).json({

status: "success",

message: "Login Sucess",

role: "hospital",

id: hospital.hospitalId,

});

} else {

res.json({ message: "Invalid Password" });

}

} else {

res.json({ message: "Invalid Email" });

}

};

const hospitalSignOutController = async (req, res) => {};

const fetchHospitalProfileController = async (req, res) => {

const { hospitalId } = req.query;

const hospital = await Hospital.findOne({ hospitalId }).lean();

if (!hospitalId) {

const response = {

status: "failed",

message: "insufficient information",

};

res.status(400).json({

response,

});

}

// handles when hospital does not exist

else if (!hospital) {

const response = {

status: "failed",

message: "hospital does not exist",

};

res.status(404).json({

response,

});

} else {

// removing password from mongodb response

const { password: rmPass, ...hospitalData } = hospital;

const response = {

status: "success",

message: "fetched hospital information successfully",

data: {

...hospitalData,

},

};

res.status(200).json({

response,

});

}

};

const hospitalCreatePatientController = async (req, res) => {

// destructering data from request body

const {

name,

email,

password,

gender,

dob,

address,

state,

phone,

hospitalId,

wallet: patientWalletAddress,

} = req.body;

const hospital = await Hospital.findOne({ hospitalId });

// handles when required data is not passed to the endpoint

if (

!name ||

!email ||

!password ||

!gender ||

!dob ||

!address ||

!state ||

!phone ||

!hospitalId ||

!patientWalletAddress

) {

const response = {

status: "failed",

message: "insufficient information",

};

res.status(400).json({

response,

});

}

// handles when hospital does not exist

else if (!hospital) {

const response = {

status: "failed",

message: "hospital does not exist",

};

res.status(404).json({

response,

});

} else {

const patientId = uuidv4();

const hospitalWalletAddress = hospital.wallet;

createPatientService(hospitalWalletAddress, patientWalletAddress, patientId)

.then(async (response) => {

if (response.status != "success") {

console.log(response);

res.status(404).json(response);

}

const patient = new Patient({

patientId,

name,

email,

password,

gender,

dob,

address,

state,

phone,

wallet: patientWalletAddress,

});

const usedAddress = new UsedAddress({ address: patientWalletAddress });

await usedAddress.save();

const result = await patient.save();

const { transactionHash } = response.data;

// removing password from mongodb response

const { password: rmPass, ...patientData } = result.\_doc;

const responseObj = {

status: "success",

message: "created patient successfully",

data: {

transactionHash,

...patientData,

},

};

res.status(200).json(responseObj);

})

.catch((err) => {

console.log(err);

const response = {

status: "failed",

message: "Internal error",

data: {

error: err.message,

},

};

res.status(500).json({

response,

});

});

}

};

const hospitalUpdatePatientMedicalReportController = async (req, res) => {

// destructering data from request body

const { hospitalId, patientId, patientReport } = req.body;

const patient = await Patient.findOne({ patientId });

const hospital = await Hospital.findOne({ hospitalId });

// handles when required data is not passed to the endpoint

if (

!patientId ||

!hospitalId ||

!patientReport ||

!patientReport.dateOfVisit ||

!patientReport.causeOfVisit ||

!patientReport.condition ||

!patientReport.description ||

!patientReport.doctor ||

!patientReport.medication

) {

const response = {

status: "failed",

message: "insufficient information",

};

res.status(400).json({

response,

});

}

// handles when patient does not exist

else if (!patient) {

const response = {

status: "failed",

message: "patient does not exist",

};

res.status(404).json({

response,

});

}

// handles when hospital does not exist

else if (!hospital) {

const response = {

status: "failed",

message: "hospital does not exist",

};

res.status(404).json({

response,

});

} else {

const hospitalWalletAddress = hospital.wallet;

const patientWalletAddress = patient.wallet;

patientReport.hospitalId = hospitalId;

patientReport.reportId = uuidv4();

patientReport.hospitalName = hospital.name;

updatePatientReportService(

hospitalWalletAddress,

patientWalletAddress,

patientReport

)

.then(async (response) => {

if (response.status != "success") {

console.log(response);

res.status(404).json(response);

} else {

res.status(200).json(response);

}

})

.catch((err) => {

console.log(err);

const response = {

status: "failed",

message: "Internal error",

data: {

error: err.message,

},

};

res.status(500).json({

response,

});

});

}

};

const authorizeHospitalController = async (req, res) => {

// destructering data from request body

const { hospitalId, patientId, hospitalToBeAuthorizedId } = req.body;

const patient = await Patient.findOne({ patientId });

const hospital = await Hospital.findOne({ hospitalId });

const hospitalToBeAuthorized = await Hospital.findOne({

hospitalId: hospitalToBeAuthorizedId,

});

// handles when required data is not passed to the endpoint

if (!patientId || !hospitalId || !hospitalToBeAuthorizedId) {

const response = {

status: "failed",

message: "insufficient information",

};

res.status(400).json({

response,

});

}

// handles when patient does not exist

else if (!patient) {

const response = {

status: "failed",

message: "patient does not exist",

};

res.status(404).json({

response,

});

}

// handles when hospital does not exist

else if (!hospital) {

const response = {

status: "failed",

message: "hospital does not exist",

};

res.status(404).json({

response,

});

}

// handles when hospital to be authorized does not exist

else if (!hospitalToBeAuthorized) {

const response = {

status: "failed",

message: "cannot authorize hospital which does not exist",

};

res.status(404).json({

response,

});

} else {

// setting wallet addresses fetched from mongodb

const patientWalletAddress = patient.wallet;

const hospitalWalletAddress = hospital.wallet;

const hospitalAddressToBeAuthorized = hospitalToBeAuthorized.wallet;

// authorizing hospital

addAuthorizedHospitalService(

hospitalWalletAddress,

patientWalletAddress,

hospitalAddressToBeAuthorized

)

.then((response) => {

if (response.status != "success") {

res.status(404).json(response);

} else {

res.status(200).json(response);

}

})

.catch((err) => {

console.log(err);

const response = {

status: "failed",

message: "Internal error",

data: {

error: err.message,

},

};

res.status(500).json({

response,

});

});

}

};

const authorizeInsuranceCompanyController = async (req, res) => {

// destructering data from request body

const { hospitalId, patientId, insuranceCompanyToBeAuthorizedId } = req.body;

const patient = await Patient.findOne({ patientId });

const hospital = await Hospital.findOne({ hospitalId });

const insuranceCompanyToBeAuthorized = await Insurance.findOne({

insuranceCompanyId: insuranceCompanyToBeAuthorizedId,

});

// handles when required data is not passed to the endpoint

if (!patientId || !hospitalId || !insuranceCompanyToBeAuthorizedId) {

const response = {

status: "failed",

message: "insufficient information",

};

res.status(400).json({

response,

});

}

// handles when patient does not exist

else if (!patient) {

const response = {

status: "failed",

message: "patient does not exist",

};

res.status(404).json({

response,

});

}

// handles when hospital does not exist

else if (!hospital) {

const response = {

status: "failed",

message: "hospital does not exist",

};

res.status(404).json({

response,

});

}

// handles when insurance company to be authorized does not exist

else if (!insuranceCompanyToBeAuthorized) {

const response = {

status: "failed",

message: "cannot authorize insurance company which does not exist",

};

res.status(404).json({

response,

});

} else {

// setting wallet addresses fetched from mongodb

const patientWalletAddress = patient.wallet;

const hospitalWalletAddress = hospital.wallet;

const insuranceCompanyAddressToBeAuthorized =

insuranceCompanyToBeAuthorized.wallet;

// authorizing hospital

addAuthorizedInsuranceCompanyService(

hospitalWalletAddress,

patientWalletAddress,

insuranceCompanyAddressToBeAuthorized

)

.then((response) => {

if (response.status != "success") {

console.log(response);

res.status(404).json(response);

} else {

res.status(200).json(response);

}

})

.catch((err) => {

console.log(err);

const response = {

status: "failed",

message: "Internal error",

data: {

error: err.message,

},

};

res.status(500).json({

response,

});

});

}

};

const fetchUnAuthorizedHospitalsController = async (req, res) => {

const { patientId, hospitalId } = req.query;

const patient = await Patient.findOne({ patientId }).lean();

const hospital = await Hospital.findOne({ hospitalId }).lean();

try {

if (!patientId || !hospitalId) {

const response = {

status: "failed",

message: "insufficient information",

};

res.status(400).json(response);

}

// handles when patient does not exist

else if (!patient) {

const response = {

status: "failed",

message: "patient does not exist",

};

res.status(404).json({

response,

});

} else if (!hospital) {

const response = {

status: "failed",

message: "unknown sender",

};

res.status(400).json(response);

} else {

fetchAuthorizedHospitalsService(patient.wallet, hospital.wallet).then(

async (response) => {

if (response.status != "success") {

res.status(404).json(response);

} else {

const authorizedHospitalsArray = response.data.authorizedHospitals;

const hospitals = await Hospital.find({}, { password: 0 });

const unauthorizedHospitals = hospitals.filter(

(hospital) => !authorizedHospitalsArray.includes(hospital.wallet)

);

response.data.authorizedHospitals = undefined;

response.data.unauthorizedHospitals = unauthorizedHospitals;

res.status(200).json(response);

}

}

);

}

} catch (error) {

console.log(error);

const response = {

status: "failed",

message: "Internal Server Error",

};

res.status(500).json(response);

}

};

const fetchUnAuthorizedInsurancesController = async (req, res) => {

const { patientId, hospitalId } = req.query;

const patient = await Patient.findOne({ patientId }).lean();

const hospital = await Hospital.findOne({ hospitalId }).lean();

try {

if (!patientId || !hospitalId) {

const response = {

status: "failed",

message: "insufficient information",

};

res.status(400).json(response);

}

// handles when patient does not exist

else if (!patient) {

const response = {

status: "failed",

message: "patient does not exist",

};

res.status(404).json({

response,

});

} else if (!hospital) {

const response = {

status: "failed",

message: "unknown sender",

};

res.status(400).json(response);

} else {

fetchAuthorizedInsuranceCompaniesService(

patient.wallet,

hospital.wallet

).then(async (response) => {

if (response.status != "success") {

res.status(404).json(response);

} else {

const authorizedInsuranceCompaniesArray =

response.data.authorizedInsuranceCompanies;

const insuranceCompanies = await Insurance.find({}, { password: 0 });

const unAuthorizedInsuranceCompanies = insuranceCompanies.filter(

(insurance) =>

!authorizedInsuranceCompaniesArray.includes(insurance.wallet)

);

response.data.authorizedInsuranceCompanies = undefined;

response.data.unAuthorizedInsuranceCompanies =

unAuthorizedInsuranceCompanies;

res.status(200).json(response);

}

});

}

} catch (error) {

console.log(error);

const response = {

status: "failed",

message: "Internal Server Error",

};

res.status(500).json(response);

}

};

const fetchAuthorizedPatientsController = async (req, res) => {

const { hospitalId } = req.query;

const hospital = await Hospital.findOne({ hospitalId }).lean();

try {

// handles when required data is not passed to the endpoint

if (!hospitalId) {

const response = {

status: "failed",

message: "insufficient information",

};

res.status(400).json({

response,

});

}

// handles when hospital does not exist

else if (!hospital) {

const response = {

status: "failed",

message: "hospital does not exist",

};

res.status(404).json({

response,

});

} else {

fetchAllPatientsService(hospital.wallet)

.then(async (response) => {

// console.log(response.data.patients[0]['authorizedHospitals']);

const patientsArray = response.data.patients;

console.log(response);

const patientIdsArray = await Promise.all(

patientsArray.map((patient) => {

if (patient["authorizedHospitals"].includes(hospital.wallet))

return patient["patientId"];

else return null;

})

);

// removing null values

const filteredPatientIdsArray = patientIdsArray.filter(

(id) => id !== null

);

const patientDetailsArray = await Promise.all(

filteredPatientIdsArray.map(async (id) => {

const patient = await Patient.findOne({ patientId: id });

return patient;

})

);

const filteredPatientDetailsArray = patientDetailsArray.filter(

(patient) => patient !== null

);

response.data.patients = undefined;

response.data.patients = filteredPatientDetailsArray;

res.send(response);

})

.catch((err) => {

const response = {

status: "failed",

message: "Internal error",

data: {

error: err.message,

},

};

res.status(500).json({

response,

});

});

}

} catch (error) {}

};

module.exports = {

hospitalSignInController,

hospitalSignOutController,

fetchHospitalProfileController,

hospitalCreatePatientController,

hospitalUpdatePatientMedicalReportController,

authorizeHospitalController,

authorizeInsuranceCompanyController,

fetchUnAuthorizedHospitalsController,

fetchUnAuthorizedInsurancesController,

fetchAuthorizedPatientsController,

};

**3 Insurance Provider**

const Hospital = require("../models/hospitalModel");

const Insurance = require("../models/insuranceModel");

const Patient = require("../models/patientModel");

const {

fetchAllPatientsService

} = require("../services/blockchain/blockchainService");

const insuranceSignInController = async (req, res) => {

const { email, password } = req.body;

const insurance = await Insurance.findOne({ email });

if (insurance) {

if (password === insurance.password) {

res.status(200).json({

status: "success",

message: "Login Sucess",

role: "insurance",

id: insurance.insuranceCompanyId,

});

} else {

res.json({ message: "Invalid Password" });

}

} else {

res.json({ message: "Invalid Email" });

}

};

const insuranceSignOutController = async (req, res) => {};

const fetchInsuranceCompanyProfileController = async (req, res) => {

const { insuranceCompanyId } = req.query;

const insuranceCompany = await Insurance.findOne({ insuranceCompanyId }).lean();

if (!insuranceCompanyId) {

const response = {

status: "failed",

message: "insufficient information",

};

res.status(400).json({

response,

});

}

// handles when hospital does not exist

else if (!insuranceCompany) {

const response = {

status: "failed",

message: "insurance company does not exist",

};

res.status(404).json({

response,

});

} else {

// removing password from mongodb response

const { password: rmPass, ...insuranceCompanyData } = insuranceCompany;

const response = {

status: "success",

message: "fetched insurance company information successfully",

data: {

...insuranceCompanyData,

},

};

res.status(200).json({

response,

});

}

};

const insuranceViewPolicyClaimsController = async (req, res) => {};

const insuranceViewPolicyClaimsDataController = async (req, res) => {};

const viewPolicyClaimsDataController = async (req, res) => {};

const insuranceRejectController = async (req, res) => {};

const insuranceAcceptController = async (req, res) => {};

const insuranceViewPolicyController = async (req, res) => {};

const fetchAuthorizedPatientsController = async (req, res) => {

const { insuranceCompanyId } = req.query;

console.log(req.query);

const insurance = await Insurance.findOne({ insuranceCompanyId }).lean();

try {

// handles when required data is not passed to the endpoint

if (!insuranceCompanyId) {

const response = {

status: "failed",

message: "insufficient information",

};

res.status(400).json({

response,

});

}

// handles when insurance does not exist

else if (!insurance) {

const response = {

status: "failed",

message: "insurance company does not exist",

};

res.status(404).json({

response,

});

}

else {

fetchAllPatientsService(insurance.wallet).then(async (response) => {

// console.log(response.data.patients[0]['authorizedInsuranceCompanies']);

const patientsArray = response.data.patients;

const patientIdsArray = await Promise.all(

patientsArray.map((patient) => {

if (patient["authorizedInsuranceCompanies"].includes(insurance.wallet))

return patient["patientId"];

else return null;

})

);

// removing null values

const filteredPatientIdsArray = patientIdsArray.filter(

(id) => id !== null

);

const patientDetailsArray = await Promise.all(

filteredPatientIdsArray.map(async (id)=> {

const patient = await Patient.findOne({patientId: id})

return patient;

})

)

response.data.patients = undefined;

response.data.patients = patientDetailsArray;

res.send(response);

});

}

} catch (error) {}

};

module.exports = {

insuranceAcceptController,

insuranceRejectController,

insuranceViewPolicyController,

insuranceAcceptController,

fetchInsuranceCompanyProfileController,

insuranceSignInController,

insuranceSignOutController,

insuranceViewPolicyClaimsController,

insuranceViewPolicyClaimsDataController,

viewPolicyClaimsDataController,

fetchAuthorizedPatientsController

};

**4 Patient Management**

const Patient = require("../models/patientModel");

const PDFDocument = require("pdfkit");

const { generateMedicalReport } = require("../utils/generateMedicalReport");

const patientSignOutController = async (req, res) => {};

const patientSignInController = async (req, res) => {

const { email, password } = req.body;

const patient = await Patient.findOne({ email });

if (patient) {

if (password === patient.password) {

res.status(200).json({

status: "success",

message: "Login Sucess",

role: "patient",

id: patient.patientId,

});

} else {

res.status(400).json({ status: "failed", message: "Invalid Password" });

}

} else {

res.status(400).json({ status: "failed", message: "Invalid Email" });

}

};

const patientViewProfileController = async (req, res) => {};

const patientViewReportController = async (req, res) => {};

const patientGenerateMedicalReportController = async (req, res) => {

const doc = new PDFDocument();

generateMedicalReport(req.body, doc);

res.setHeader("Content-Type", "application/pdf");

// res.setHeader("Content-Disposition", "attachment; filename=medical-report.pdf");

// Pipe the PDF document to the response

doc.pipe(res);

// Finalize the PDF document

doc.end();

};

const patientViewHospitalAccessController = async (req, res) => {};

const patientViewInsuranceAccessController = async (req, res) => {};

const patientViewpatientAccessController = async (req, res) => {};

module.exports = {

patientSignInController,

patientSignOutController,

patientViewProfileController,

patientViewReportController,

patientGenerateMedicalReportController,

patientViewpatientAccessController,

patientViewInsuranceAccessController,

patientViewHospitalAccessController,

};

* **Solidity File- For adding and viewing blockchain smart contract data**

// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.9;

contract MedicalRecords {

address public admin;

struct Patient {

string patientId;

address[] authorizedHospitals;

address[] authorizedInsuranceCompanies;

string[] patientData;

}

struct InsuranceCompany {

string insuranceProviderId;

address[] patientAddresses;

}

struct Hospital {

string hospitalId;

address[] patientAddresses;

}

// mappings to store patient, insurance company and hospitals

mapping(address => Patient) public patients;

mapping(address => InsuranceCompany) public insuranceCompanies;

mapping(address => Hospital) public hospitals;

// arrays to store addresses

address[] public hospitalAddresses;

address[] public patientAddresses;

address[] public insuranceCompanyAddresses;

// restricts to admin

modifier onlyAdmin() {

require(msg.sender == admin, "Only admin can perform this action.");

\_;

}

// restricts to hospital

modifier onlyHospital() {

require(

bytes(hospitals[msg.sender].hospitalId).length != 0,

"Only a hospital can perform this action."

);

\_;

}

// Modifier to restrict access to authorized hospitals

modifier onlyAuthorizedHospital(address patientAddress) {

bool isAuthorized = false;

for (

uint256 i = 0;

i < patients[patientAddress].authorizedHospitals.length;

i++

) {

if (patients[patientAddress].authorizedHospitals[i] == msg.sender) {

isAuthorized = true;

break;

}

}

require(

isAuthorized,

"Only authorized hospitals can perform this action."

);

\_;

}

constructor() {

admin = msg.sender;

}

// Function to view the admin address

function getAdminAddress() public view returns (address) {

return admin;

}

// function to create hospital by admin

function createHospital(

address hospitalAddress,

string memory hospitalId

) public onlyAdmin {

hospitals[hospitalAddress].hospitalId = hospitalId;

hospitals[hospitalAddress].patientAddresses = new address[](0);

hospitalAddresses.push(hospitalAddress);

}

// function to create insurance company by admin

function createInsuranceCompany(

address insuranceCompanyAddress,

string memory insuranceProviderId

) public onlyAdmin {

insuranceCompanies[insuranceCompanyAddress]

.insuranceProviderId = insuranceProviderId;

insuranceCompanies[insuranceCompanyAddress]

.patientAddresses = new address[](0);

insuranceCompanyAddresses.push(insuranceCompanyAddress);

}

function addPatient(

address patientAddress,

string memory patientId

) public onlyHospital {

patients[patientAddress].patientId = patientId;

patients[patientAddress].authorizedHospitals.push(msg.sender);

// adding patient to the patient list of the hospital

hospitals[msg.sender].patientAddresses.push(patientAddress);

patientAddresses.push(patientAddress);

}

// get all hospital details

function getAllHospitals() public view returns (Hospital[] memory) {

Hospital[] memory allHospitals = new Hospital[](

hospitalAddresses.length

);

for (uint256 i = 0; i < hospitalAddresses.length; i++) {

allHospitals[i] = hospitals[hospitalAddresses[i]];

}

return allHospitals;

}

function getAllPatients() public view returns (Patient[] memory) {

Patient[] memory allPatients = new Patient[](patientAddresses.length);

for (uint256 i = 0; i < patientAddresses.length; i++) {

allPatients[i] = patients[patientAddresses[i]];

}

return allPatients;

}

function getAllInsuranceCompanies()

public

view

returns (InsuranceCompany[] memory)

{

InsuranceCompany[]

memory allInsuranceCompanies = new InsuranceCompany[](

insuranceCompanyAddresses.length

);

for (uint256 i = 0; i < insuranceCompanyAddresses.length; i++) {

allInsuranceCompanies[i] = insuranceCompanies[

insuranceCompanyAddresses[i]

];

}

return allInsuranceCompanies;

}

function getPatientInfo(

address patientAddress

) public view returns (Patient memory) {

return patients[patientAddress];

}

function getHospitalInfo(

address hospitalAddress

) public view returns (Hospital memory) {

return hospitals[hospitalAddress];

}

function getInsuranceCompanyInfo(

address insuranceCompanyAddress

) public view returns (InsuranceCompany memory) {

return insuranceCompanies[insuranceCompanyAddress];

}

// update patient data

// Function to update the patientData of a specific patient

function updatePatientData(

address patientAddress,

string memory newPatientData

) public onlyAuthorizedHospital(patientAddress) {

patients[patientAddress].patientData.push(newPatientData);

}

function getAuthorizedHospitals(

address patientAddress

) public view returns (address[] memory) {

return patients[patientAddress].authorizedHospitals;

}

// Function to get the addresses of authorized insurance companies for a patient

function getAuthorizedInsuranceCompanies(

address patientAddress

) public view returns (address[] memory) {

return patients[patientAddress].authorizedInsuranceCompanies;

}

// authorized hospital can push a insurance company to the authorized insurance companies list

function addAuthorizedInsuranceCompany(

address patientAddress,

address insuranceCompanyAddress

) public onlyAuthorizedHospital(patientAddress) {

patients[patientAddress].authorizedInsuranceCompanies.push(

insuranceCompanyAddress

);

}

// authorized hospital can push a another hospital to the authorized hospitals list

function addAuthorizedHospital(

address patientAddress,

address hospitalAddress

) public onlyAuthorizedHospital(patientAddress) {

patients[patientAddress].authorizedHospitals.push(hospitalAddress);

}

}

* **Functions for interacting with backend blockchain smart contract**

// configuring environment

require("dotenv").config();

const Web3 = require("web3");

// importing artifacts

const contractArtifact = require("../../../build/contracts/MedicalRecords.json");

// address of deployed contract

const contractAddress = process.env.MEDICAL\_RECORD\_CONTACT\_ADDR;

// connection to ganache network

const ganacheConnectionUrl =

"http://" + process.env.GANACHE\_HOST + ":" + process.env.GANACHE\_PORT;

const web3 = new Web3(ganacheConnectionUrl);

const contract = new web3.eth.Contract(contractArtifact.abi, contractAddress);

// fetch admin address

const fetchAdminAddressService = async () => {

try {

const adminAddress = await contract.methods.getAdminAddress().call();

// console.log("Admin address:", adminAddress);

const response = {

status: "success",

message: "fetched admin address",

data: {

adminAddress,

},

};

return response;

} catch (error) {

// console.error("Error calling getAdminAddress:", error);

const response = {

status: "failed",

message: "error fetching admin address",

data: {

error: error.message,

},

};

return response;

}

};

// create a hospital

const createHospitalService = async (

adminAccount,

hospitalAddress,

hospitalId

) => {

try {

// Call the createHospital function

const transaction = contract.methods.createHospital(

hospitalAddress,

hospitalId

);

const gas = await transaction.estimateGas({ from: adminAccount });

const result = await transaction.send({ from: adminAccount, gas });

const response = {

status: "success",

message: "created hospital successfully",

data: {

transactionHash: result.transactionHash,

},

};

return response;

} catch (error) {

// console.error("Error calling createHospital:", error);

const response = {

status: "failed",

message: "error while creating hospital",

data: {

error: error.message,

},

};

return response;

}

};

// fetch all hospitals

const fetchAllHospitalsService = async () => {

try {

const hospitals = await contract.methods.getAllHospitals().call();

// console.log("List of hospitals:", hospitals);

const response = {

status: "success",

message: "fetched hospitals successfully",

data: {

hospitals,

},

};

return response;

} catch (error) {

// console.error("Error calling getAllHospitals:", error);

const response = {

status: "failed",

message: "error while fetching hospitals",

data: {

error: error.message,

},

};

return response;

}

};

// create a patient

const createPatientService = async (

hospitalAccount,

patientAddress,

patientId

) => {

try {

// Call the addPatient function

const transaction = contract.methods.addPatient(patientAddress, patientId);

const gas = await transaction.estimateGas({ from: hospitalAccount });

const result = await transaction.send({ from: hospitalAccount, gas });

// console.log(

// "Patient added successfully. Transaction hash:",

// result.transactionHash

// );

const response = {

status: "success",

message: "created patient successfully",

data: {

transactionHash: result.transactionHash,

},

};

return response;

} catch (error) {

// console.error("Error calling addPatient:", error);

const response = {

status: "failed",

message: "error while creating patient",

data: {

error: error.message,

},

};

return response;

}

};

// fetch all patients

const fetchAllPatientsService = async (senderWalletAddress) => {

try {

// const patients = await contract.methods.getAllPatients().call();

// console.log("List of patients:", patients);

console.log(senderWalletAddress);

const transaction = contract.methods.getAllPatients();

const gas = await transaction.estimateGas({ from: senderWalletAddress });

const patients = await transaction.call({

from: senderWalletAddress,

gas: gas,

});

console.log(gas);

const response = {

status: "success",

message: "fetched patients successfully",

data: {

patients,

},

};

return response;

} catch (error) {

// console.error("Error calling getAllPatients:", error);

const response = {

status: "failed",

message: "error while fetching patients",

data: {

error: error.message,

},

};

return response;

}

};

// create an insurance company

const createInsuranceCompanyService = async (

adminAccount,

insuranceCompanyAddress,

insuranceProviderId

) => {

try {

// Call the createInsuranceCompany function

const transaction = contract.methods.createInsuranceCompany(

insuranceCompanyAddress,

insuranceProviderId

);

const gas = await transaction.estimateGas({ from: adminAccount });

const result = await transaction.send({ from: adminAccount, gas });

const response = {

status: "success",

message: "created insurance company successfully",

data: {

transactionHash: result.transactionHash,

},

};

return response;

} catch (error) {

const response = {

status: "failed",

message: "error while creating insurance company",

data: {

error: error.message,

},

};

return response;

}

};

// fetch all insurance companies

const fetchAllInsuranceCompaniesService = async () => {

try {

const insuranceCompanies = await contract.methods

.getAllInsuranceCompanies()

.call();

// console.log("List of insurance companies:", insuranceCompanies);

const response = {

status: "success",

message: "fetched insurance companies successfully",

data: {

insuranceCompanies,

},

};

return response;

} catch (error) {

// console.error("Error calling getAllInsuranceCompanies:", error);

const response = {

status: "failed",

message: "error while fetching insurance companies",

data: {

error: error.message,

},

};

return response;

}

};

// update patient data ( newPatientData is stringified JSON)

const updatePatientReportService = async (

hospitalAddress,

patientAddress,

newPatientData

) => {

try {

// PATIENT DATA FORMAT

// const patientDataJSON = {

// content: "exists",

// data: {

// cancer: "true",

// },

// };

// Call the updatePatientData function

const transaction = contract.methods.updatePatientData(

patientAddress,

JSON.stringify(newPatientData)

);

const gas = await transaction.estimateGas({ from: hospitalAddress });

const result = await transaction.send({ from: hospitalAddress, gas });

// console.log("Patient data updated successfully!");

const response = {

status: "success",

message: "updated patient data successfully",

data: {

transactionHash: result.transactionHash,

},

};

return response;

} catch (error) {

// console.error("Error updating patient data:", error);

const response = {

status: "failed",

message: "error while updating patient data",

data: {

error: error.message,

},

};

return response;

}

};

// fetch authorized hospitals for a patient

const fetchAuthorizedHospitalsService = async (

patientAddress,

senderWalletAddress

) => {

try {

// Call the getAuthorizedHospitals function in the contract

const transaction = contract.methods.getAuthorizedHospitals(patientAddress);

const gas = await transaction.estimateGas({ from: senderWalletAddress });

const authorizedHospitals = await transaction.call({

from: senderWalletAddress,

gas: gas,

});

// console.log("Authorized hospitals:", authorizedHospitals);

const response = {

status: "success",

message: "fetched authorized hospitals successfully",

data: {

authorizedHospitals,

},

};

return response;

} catch (error) {

// console.error("Error getting authorized hospitals:", error);

const response = {

status: "failed",

message: "error while fetching authorized hospitals",

data: {

error: error.message,

},

};

return response;

}

};

// fetch authorized insurance companies for a patient

const fetchAuthorizedInsuranceCompaniesService = async (

patientAddress,

senderWalletAddress

) => {

try {

console.log(senderWalletAddress);

// Call the getAuthorizedInsuranceCompanies function in the contract

const transaction =

contract.methods.getAuthorizedInsuranceCompanies(patientAddress);

const gas = await transaction.estimateGas({ from: senderWalletAddress });

const authorizedInsuranceCompanies = await transaction.call({

from: senderWalletAddress,

gas: gas,

});

const response = {

status: "success",

message: "fetched authorized insurance companies successfully",

data: {

authorizedInsuranceCompanies,

},

};

return response;

} catch (error) {

// console.error("Error getting authorized insurance companies:", error);

const response = {

status: "failed",

message: "error while fetching authorized insurance companies",

data: {

error: error.message,

},

};

return response;

}

};

// function to authorize an insurance company for the patient

const addAuthorizedInsuranceCompanyService = async (

hospitalAddress,

patientAddress,

insuranceAddress

) => {

try {

const transaction = contract.methods.addAuthorizedInsuranceCompany(

patientAddress,

insuranceAddress

);

const gas = await transaction.estimateGas({ from: hospitalAddress });

const result = await transaction.send({ from: hospitalAddress, gas });

const response = {

status: "success",

message: "authorized insurance company successfully",

data: {

transactionHash: result.transactionHash,

},

};

return response;

} catch (error) {

// console.error('Error adding insurance company:', error);

const response = {

status: "failed",

message: "error while authorizing insurance company",

data: {

error: error.message,

},

};

return response;

}

};

// function to authorize a hospital company for the patient

const addAuthorizedHospitalService = async (

hospitalAddress,

patientAddress,

hospitalAddressToBeAuthorized

) => {

try {

const transaction = contract.methods.addAuthorizedHospital(

patientAddress,

hospitalAddressToBeAuthorized

);

const gas = await transaction.estimateGas({ from: hospitalAddress });

const result = await transaction.send({ from: hospitalAddress, gas });

const response = {

status: "success",

message: "authorized hospital successfully",

data: {

transactionHash: result.transactionHash,

},

};

return response;

} catch (error) {

const response = {

status: "failed",

message: "error while authorizing hospital",

data: {

error: error.message,

},

};

return response;

}

};

// function to retrieve all wallet addresses from ganache

const fetchWalletAddressesService = async () => {

try {

const accounts = await web3.eth.getAccounts();

const response = {

status: "success",

message: "fetched wallet addresses successfully",

data: {

addresses: accounts,

},

};

return response;

} catch (error) {

const response = {

status: "failed",

message: "error while fetching wallet addresses",

data: {

error: error.message,

},

};

return response;

}

};

// fetch medical records contract address

const fetchMedicalRecordsContractService = async () => {

try {

const blockNumber = await web3.eth.getBlockNumber();

let contractAddresses = [];

for (let i = 0; i <= blockNumber; i++) {

const block = await web3.eth.getBlock(i);

for (const txHash of block.transactions) {

const receipt = await web3.eth.getTransactionReceipt(txHash);

if (receipt.contractAddress) {

contractAddresses.push(receipt.contractAddress);

}

}

}

const response = {

status: "success",

message: "fetched medical records contract address successfully",

data: {

contractAddress: contractAddresses[contractAddresses.length - 2],

},

};

return response;

} catch (error) {

const response = {

status: "failed",

message: "error while fetching medical records contract address",

data: {

error: error.message,

},

};

return response;

}

};

// fetch information about single patient

const fetchPatientInfoService = async (patientAddress, senderWalletAddress) => {

try {

const transaction = contract.methods.getPatientInfo(patientAddress);

const gas = await transaction.estimateGas({ from: senderWalletAddress });

const patientInfo = await transaction.call({

from: senderWalletAddress,

gas: gas,

});

const response = {

status: "success",

message: "fetched patient information successfully",

data: {

patientInfo: patientInfo,

},

};

return response;

} catch (error) {

console.log(error);

const response = {

status: "failed",

message: "error while fetching patient information",

data: {

error: error.message,

},

};

return response;

}

};

// fetch information about single hospital

const fetchHospitalInfoService = async (hospitalAddress) => {

try {

const hospitalInfo = await contract.methods

.getHospitalInfo(hospitalAddress)

.call();

const response = {

status: "success",

message: "fetched hospital information successfully",

data: {

hospitalInfo: hospitalInfo,

},

};

return response;

} catch (error) {

const response = {

status: "failed",

message: "error while fetching hospital information",

data: {

error: error.message,

},

};

return response;

}

};

// fetch information about single insurance company

const fetchInsuranceCompanyInfoService = async (insuranceCompanyAddress) => {

try {

const insuranceCompanyInfo = await contract.methods

.getInsuranceCompanyInfo(insuranceCompanyAddress)

.call();

const response = {

status: "success",

message: "fetched insurance company information successfully",

data: {

insuranceCompanyInfo: insuranceCompanyInfo,

},

};

return response;

} catch (error) {

const response = {

status: "failed",

message: "error while fetching insurance company information",

data: {

error: error.message,

},

};

return response;

}

};

module.exports = {

addAuthorizedHospitalService,

addAuthorizedInsuranceCompanyService,

createHospitalService,

createInsuranceCompanyService,

createPatientService,

fetchAdminAddressService,

fetchAllHospitalsService,

fetchAllInsuranceCompaniesService,

fetchAllPatientsService,

fetchAuthorizedHospitalsService,

fetchAuthorizedInsuranceCompaniesService,

fetchInsuranceCompanyInfoService,

fetchMedicalRecordsContractService,

fetchPatientInfoService,

fetchHospitalInfoService,

fetchWalletAddressesService,

updatePatientReportService,

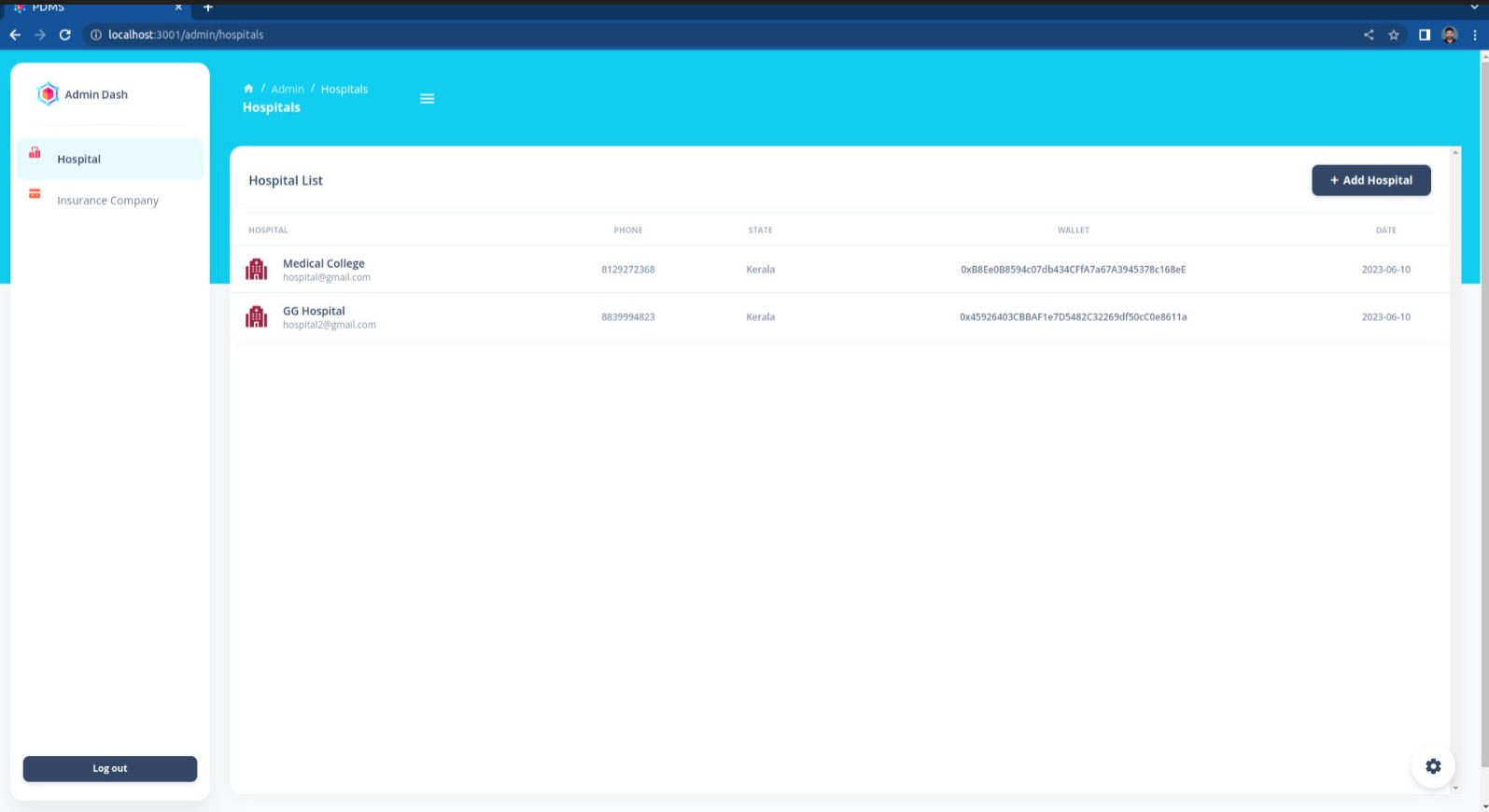
};

14.2 OUTPUT SCREENSHOTS

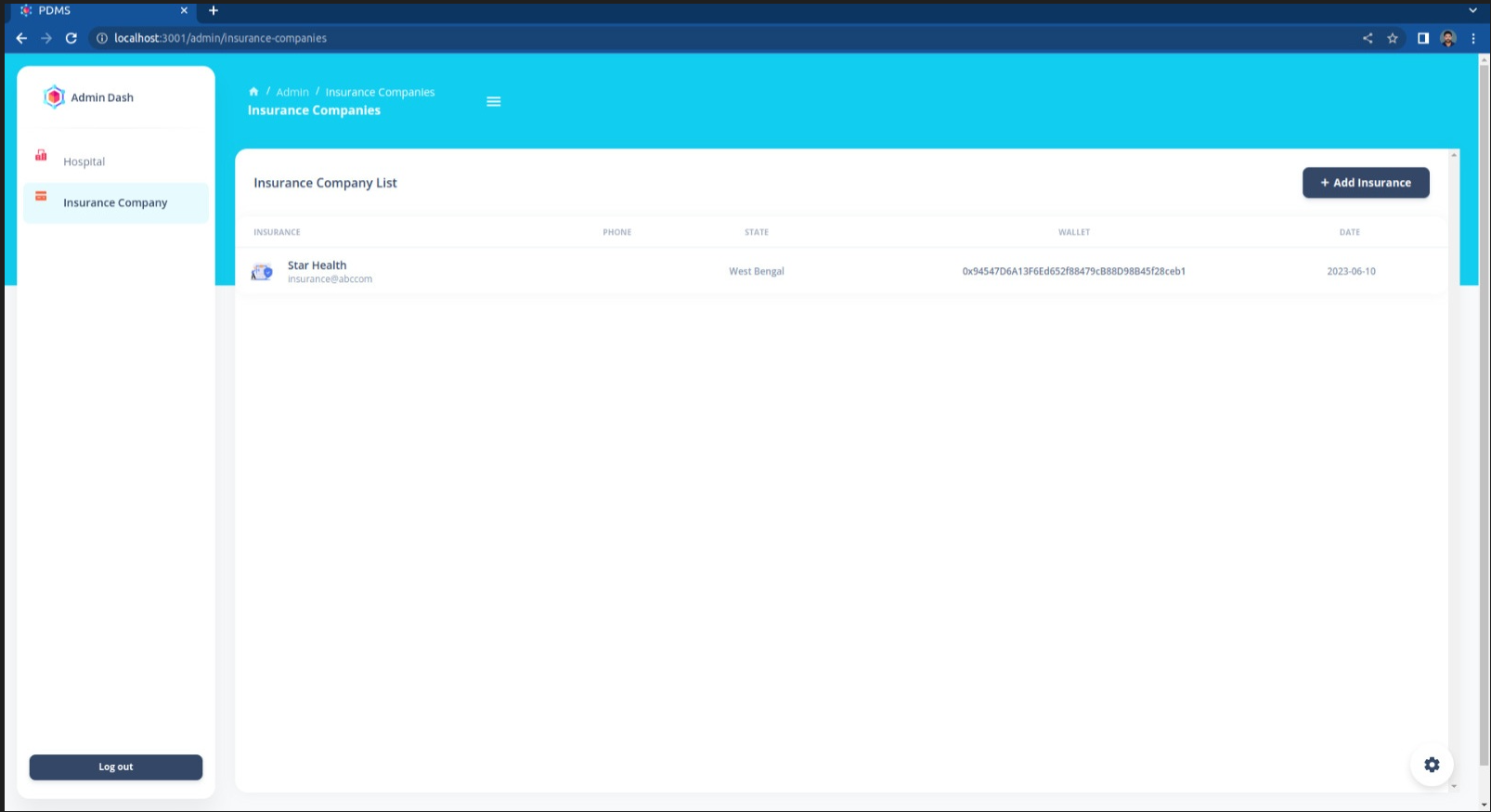
* ADMIN ADDS HOSPITAL TO THE SYSTEM



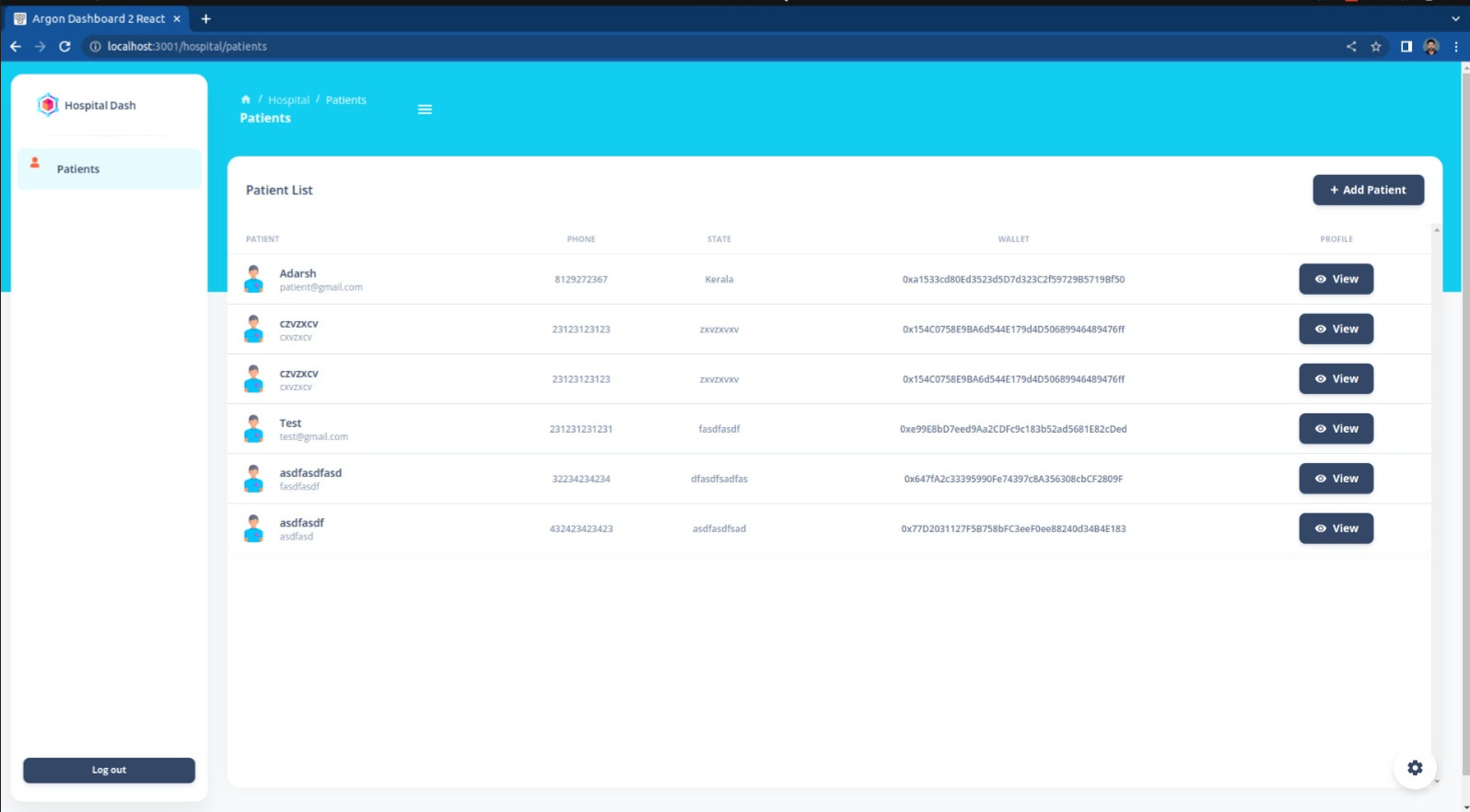
* ADMIN ADDS HOSPITAL LIST TO THE SYSTEM



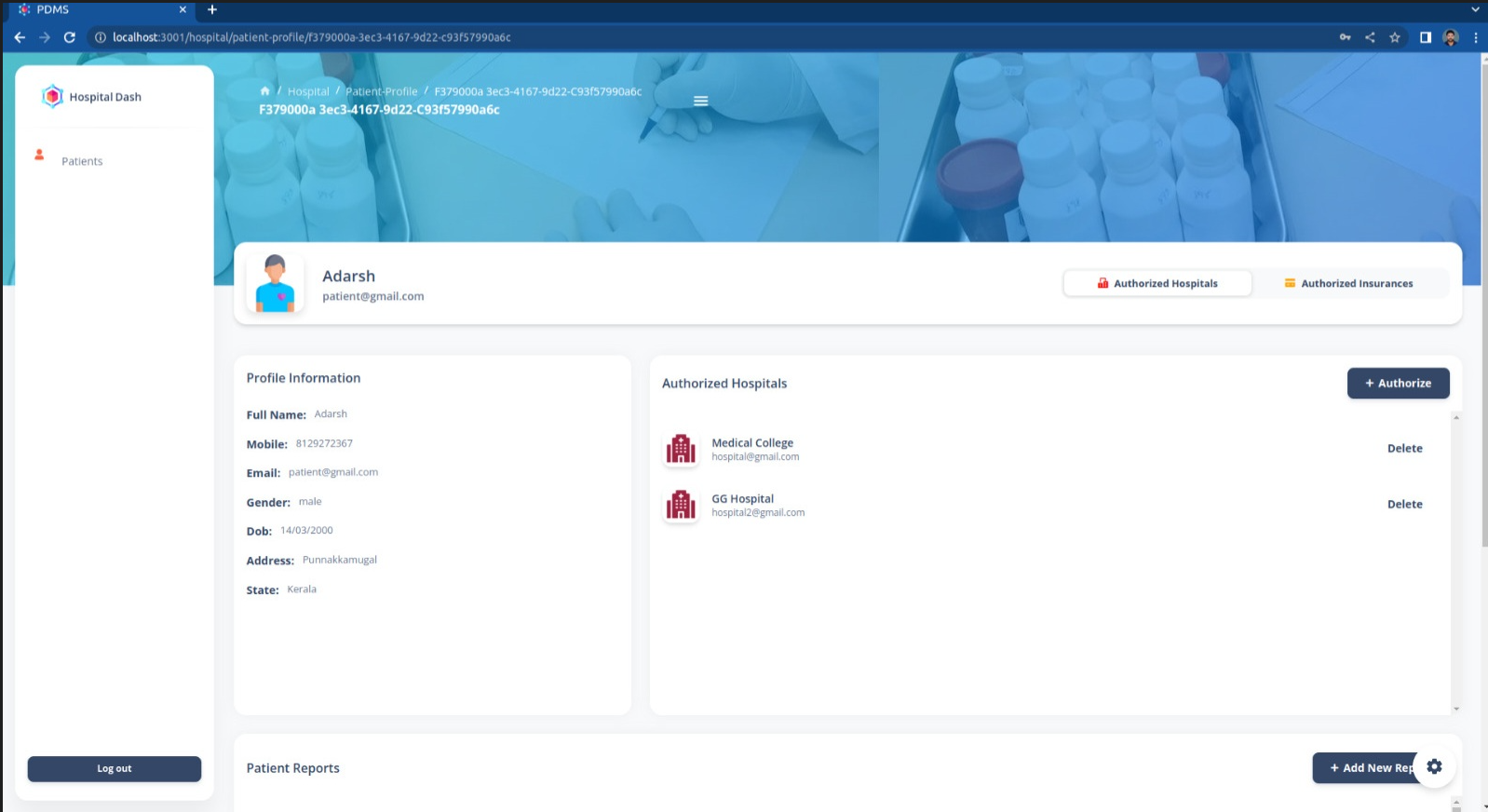
* ADMIN ADDS INSURANCE AND INSURANCE COMPANIES



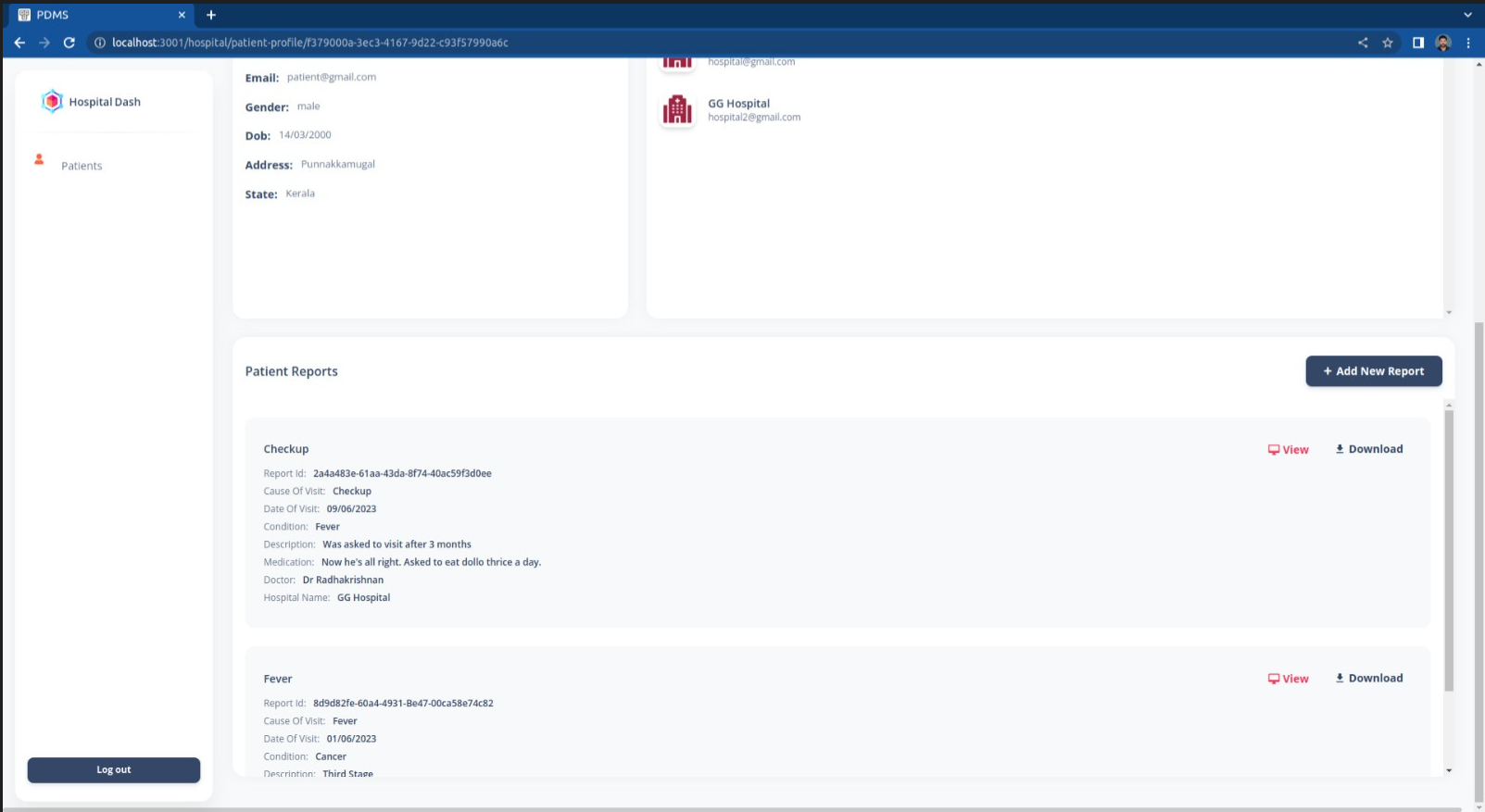
* PATIENT’S LIST IN HOSPITAL PAGE



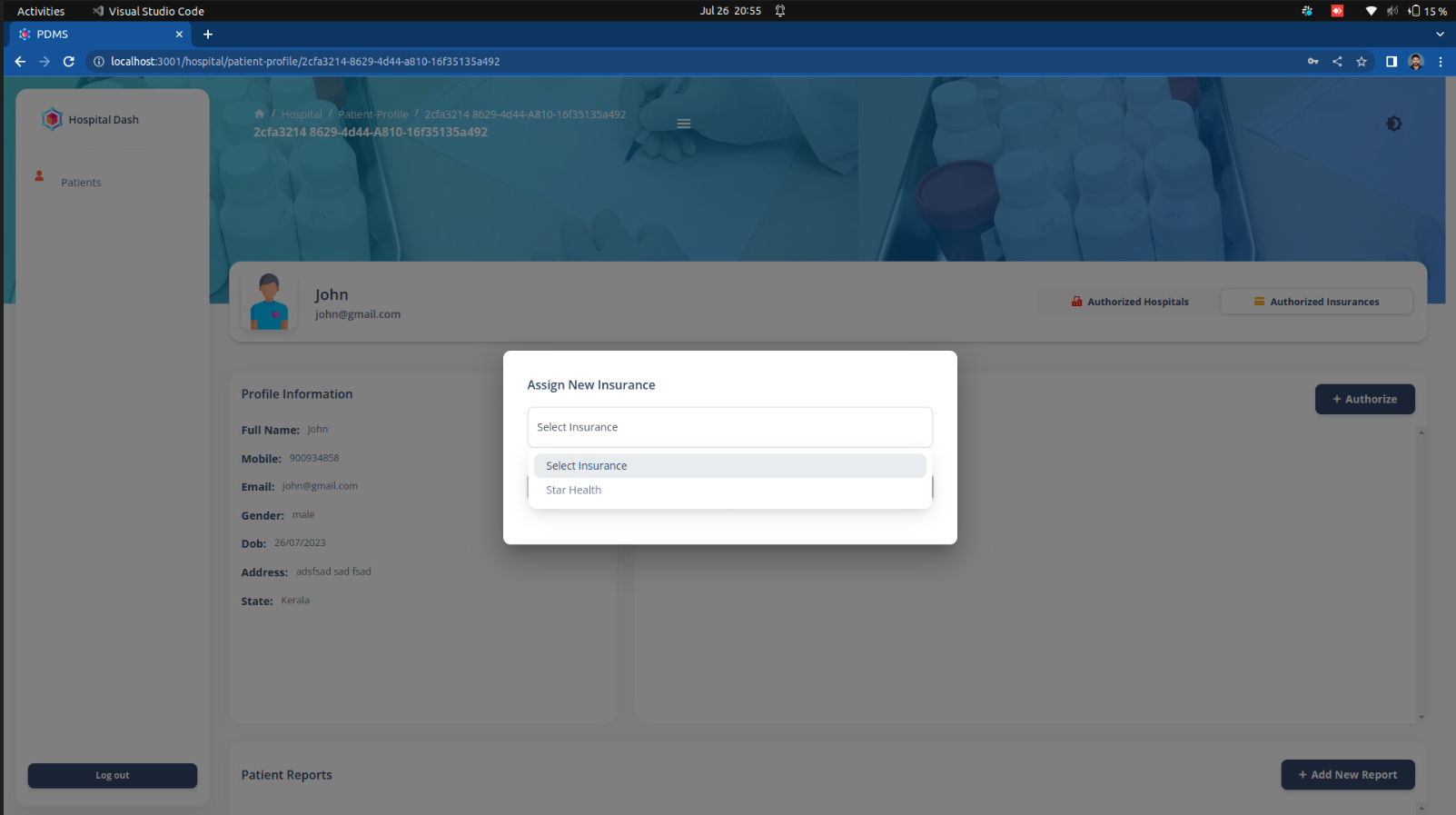
* VIEW PATIENT’S PROFILE



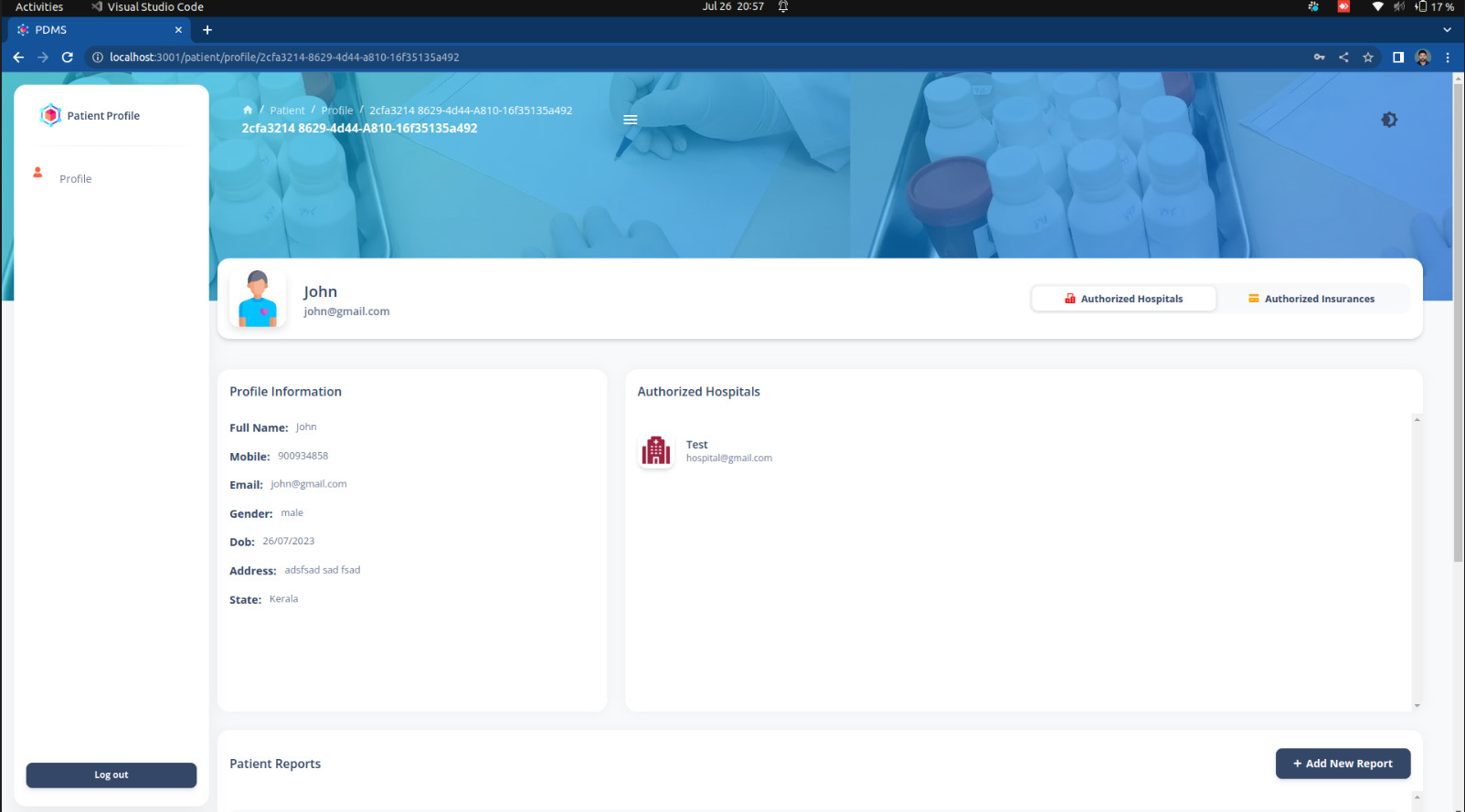
* PATIENT’S REPORT IN HOSPITAL PAGE



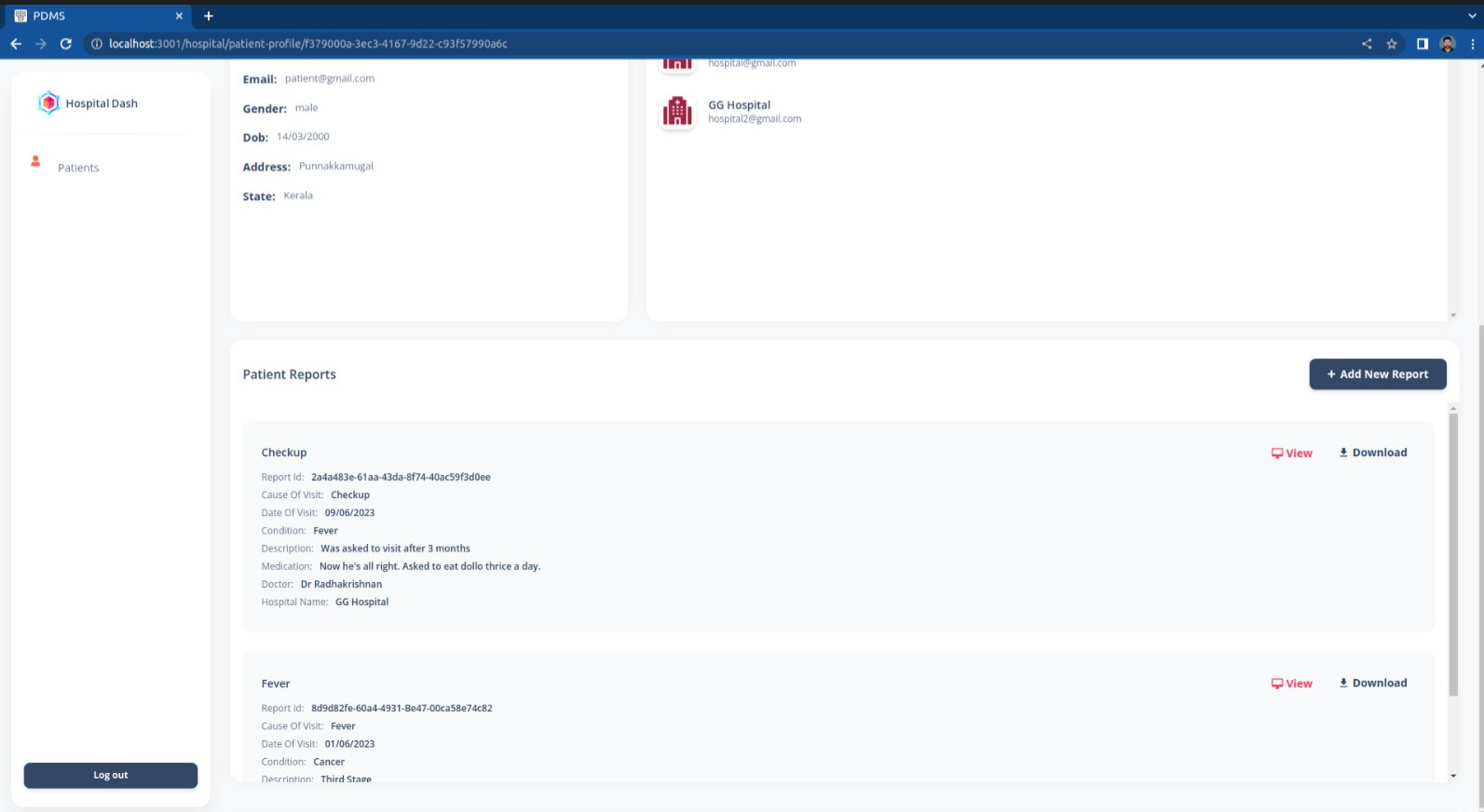
* AUTHORIZE INSURANCE TO PATIENT IN HOSPITAL PAGE



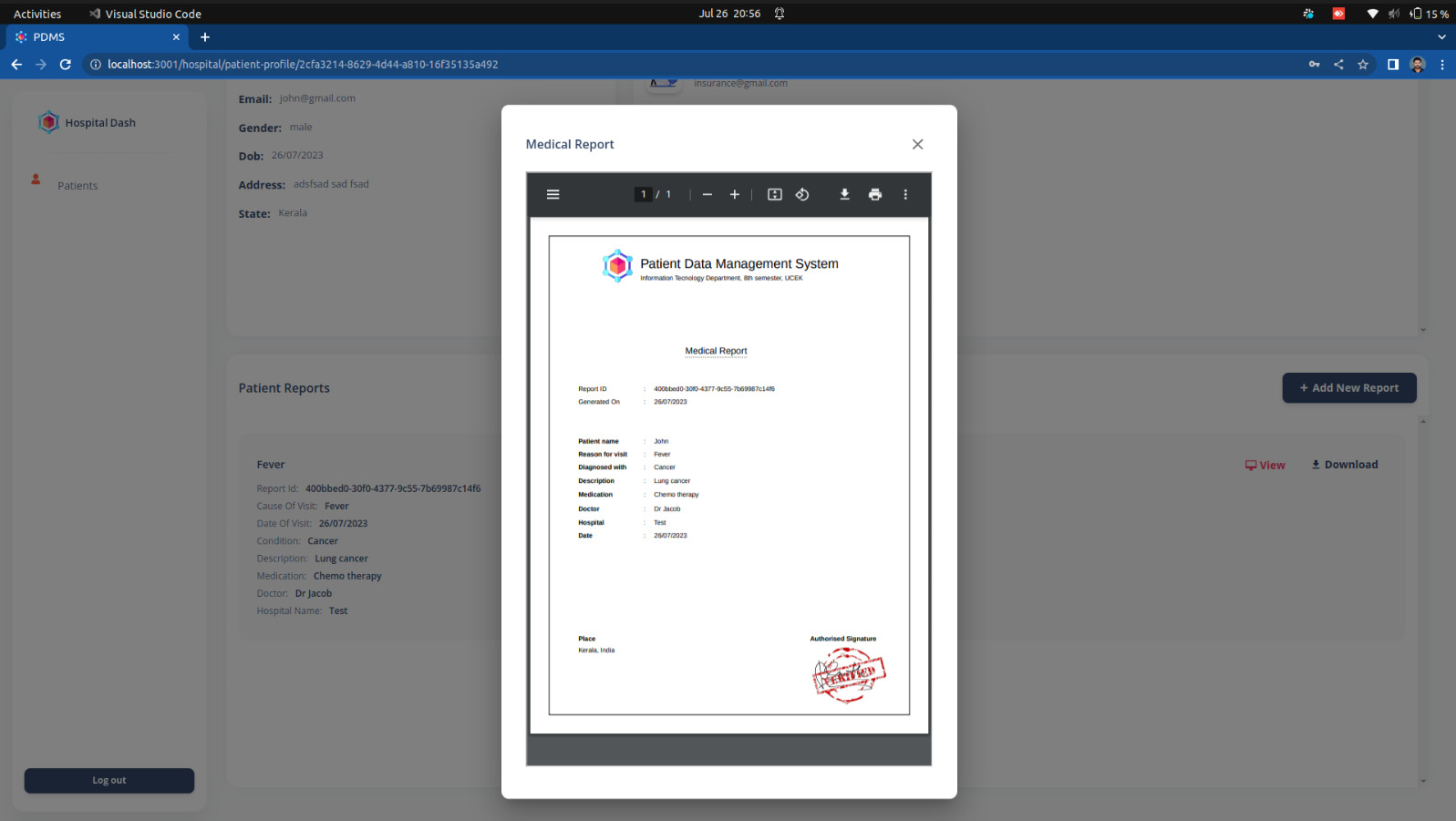
* PATIENT PROFILE CAN VIEW AUTHORIZED HOSPITAL’S AND INSURANCE COMPANIES



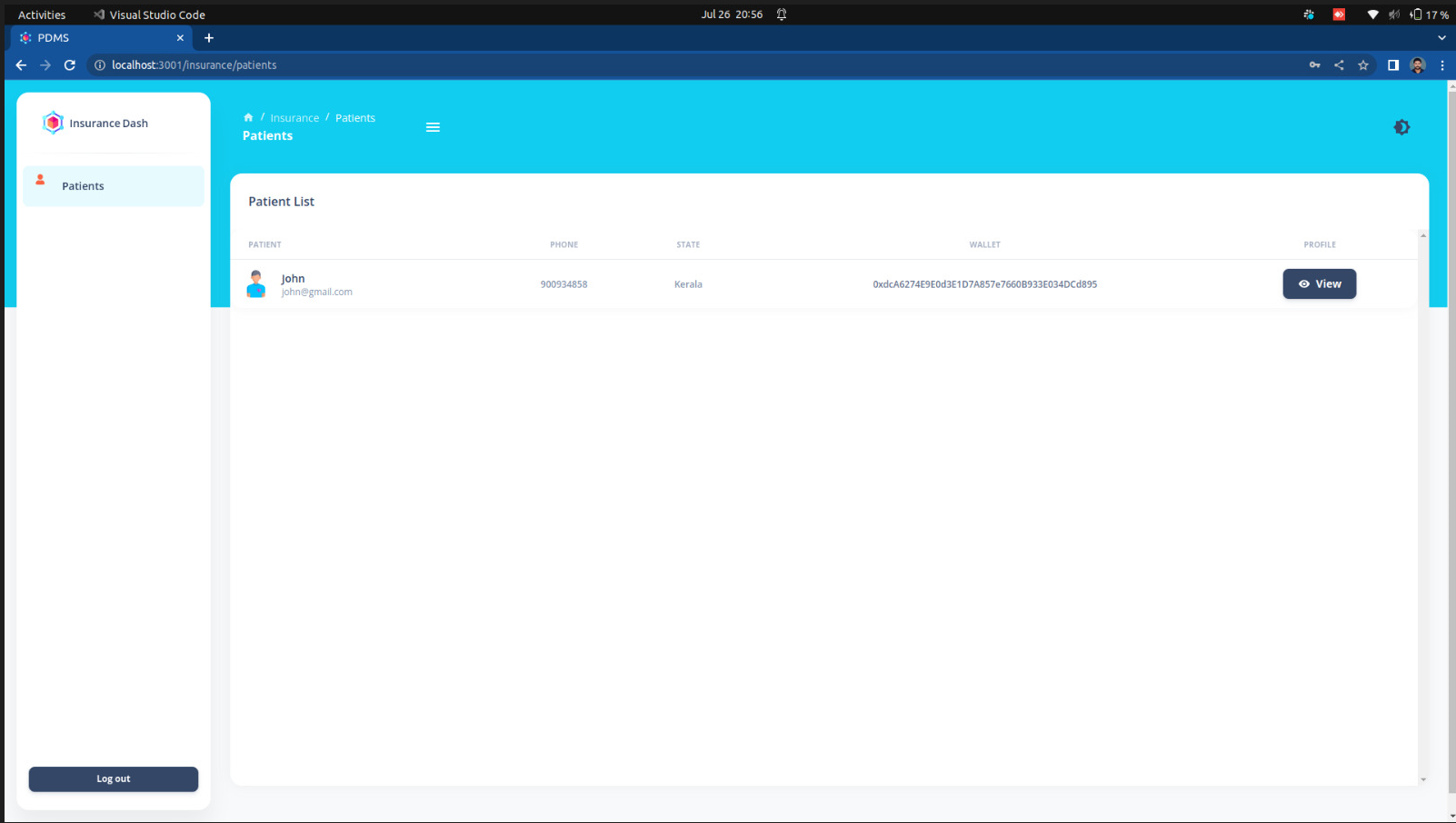
* PATIENT’S PAGE – VIEW AND SHARE REPORT BY DOWNLOADING



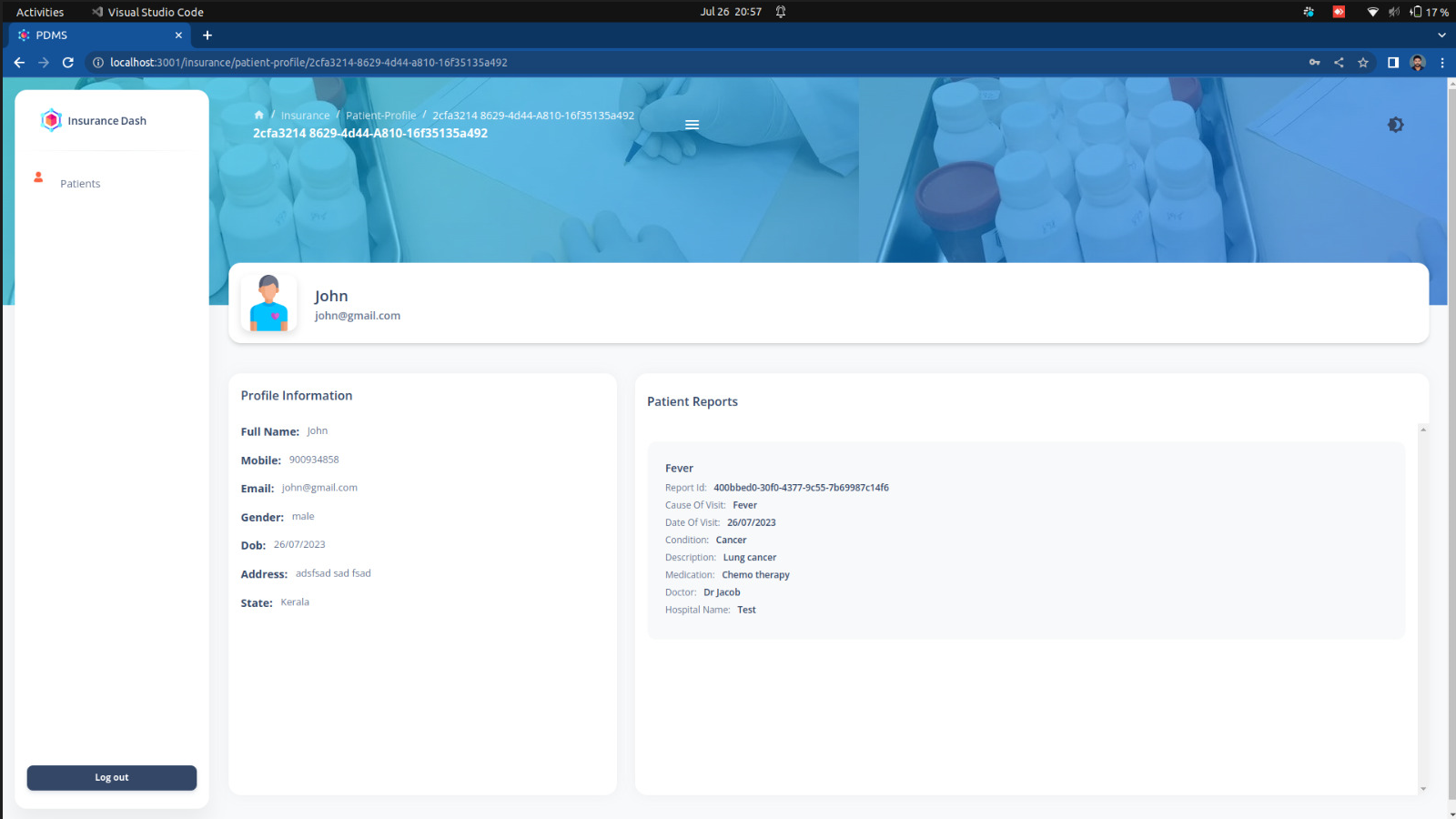
* PATIENT’S MEDICAL REPORT CAN BE SEEN IN PATIENT PAGE , HOSPITAL PAGE AND AUTHIRIZED INSURANCE PAGE



* INSURANCE VIEW AUTHORIZED PATIENTS



* INSURANCE VIEW PATIENT PROFILE AND MEDICAL REPORTS



15. REFERENCES

[1] Azaria A, Ekblaw A, Vieira T, Lippman A. Medrec: Using blockchain for medical data access and permission management. In 2016 2nd International Conference on Open and Big Data (OBD) 2016 Aug;:25–30.

[2] A. Dubovitskaya, Z. Xu, S. Ryu, M. Schumacher, and F. Wang, “Secure and Trustable Electronic Medical Records Sharing Using Blockchain,” in AMIA (American Medical Informatics Association) Annual Symposium Proceedings ,2017

[3] Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” White Paper, 2008.

[4] Wood, “Ethereum: A secure decentralised generalized transaction ledger,” Ethereum Project Yellow Paper, 2014.

[5] The Office of the Nat. Coordinator for Health InformationTechnology, “Report on health information blocking,” U.S. Department of HHS, Tech. Rep., 2015.

[6] Health Information and the Law, “Who owns medialrecords: 50 state comparison,” 2015.

[7] Lu, Z.; Liu, W.; Wang, Q.; Qu, G.; Liu, Z. A Privacy-Preserving Trust Model Based on Blockchain for VANETs. IEEE Access 2018, 6, 45655– 45664.