Exploring the Integration of Blockchain Technology in the Healthcare Sector: Opportunities, Challenges, and Implications.

Abstract

Blockchain has emerged as a disruptive technology that was able to revolutionize the field of cryptocurrencies. With its successful implementation in the aforementioned field, opportunities exist to assess the feasibility of the application of blockchain technology in different industry sectors such as supply chain management, law and real estate. The application of blockchain technology in the healthcare sector could divulge many prospects and opportunities to securely store patient data in a tamper-resistant manner. Furthermore, it would allow data to be in a decentralized manner without the implication of a central data governing authority. Furthermore, blockchain could also be used to address the current limitations of data storage address patient data privacy concerns and prevent susceptible attacks from malicious authors. Therefore, this paper proposes a novel approach to securely store patient data using blockchain technology. A novel algorithm named as Cross-Hash Validator Algorithm has been implemented to the core of the blockchain model which can be utilized to furthure increase the security of data storage and minimize data reduncdancies. Furthermore, the proposed blockchain platform will also facilitate a streamlined process of patient data consent management and the provision of medical records in an anonymous manner for business analytical purposes.

Table Of Content

Table of Figures

CHAPTER 01 : INTRODUCTION

1.1 - Introduction

Blockchain is a decentralized and distributed digital ledger technology that records transactions across multiple computers in a way that ensures the security, transparency, and immutability of the data. Today, blockchain technology has laid the foundation for its most renowned form of application, which is cryptocurrencies such as Bitcoin and Ethereum, that utilize this technology to facilitate secure and decentralized digital transactions.

In a time with the rapid growth of technology and widespread usage of the Internet, the healthcare industry is encountering escalating challenges and potential threats and dangers concerning securely managing patient medical records. The digitalization of health records and integration of connected devices has undoubtedly improved the efficiency and accessibility of medical records of patients. However, these traditional methodologies have also created possibilities for new vulnerabilities that could be exploited by malicious authors.

Blockchain technology shows promising prospects which could revolutionize the healthcare industry, specifically in securely managing and storing medical records of patients. The aforementioned technology which was pioneered with its application in cryptocurrency has been successfully adopted in many industry sectors such as supply chain management, law, and real estate. However, the current research towards the feasibility of applying blockchain technology in the healthcare industry is very limited.

Therefore, this paper aims to evaluate the feasibility of the application of blockchain technology in the healthcare sector to enhance the secure storage and efficient management of patient data. Furthermore, we propose a novel approach to securely store patient data using blockchain technology which also provides the ability for patient data consent management.

1.2 - Background & Motivation

In the digital age, security and privacy are critical aspects that every individual and organisation need to prioritize. In the early days of healthcare, much priority was not given to efficiently managing patient records pertaining to their medical history. Patient records were meticulously recorded manually in ledgers and using conventional file systems. This legacy approach had its share of drawbacks as it was nearly impossible to scale and efficiently manage a large amount of data pertaining to patients. However, with the passage of time and technological evolution manual record-keeping was slowly digitized and currently, every major aspect of the healthcare sector has been revolutionized by computerized systems.

Currently, patient data are commonly found to be stored in a centralized system and are managed by a central authority such as a hospital. However, it was evident this centralized approach to storing patient data led to redundancies and duplication of patient health records across multiple healthcare institutions. This duplication not only hampers the efficiency of data management but also poses challenges in maintaining a unified and accurate patient record.

In the existing centralized models, where patient data is stored and managed by a central authority like a hospital, duplications and inconsistencies can occur when the same patient seeks care across different healthcare institutions. The causes of inefficient management of patient data and redundancies across different data storage facilities.

With a blockchain-based system, a patient's health record becomes a unified, secure, and easily accessible ledger shared across all authorized participants in the network. Blockchain technology, while initially popularized by its association with cryptocurrencies like Bitcoin,

offers a wide range of applications beyond the financial sector. Blockchain technology has been successfully utilized in various industries like supply chain management, law, real estate, voting systems, and healthcare to ensure information integrity and security. One of the industries poised to benefit significantly from blockchain is healthcare. However, the existing research conducted in studying the feasibility of applying blockchain technology is very limited and provides us with a lot of opportunities to assess the possibility of developing a novel system using blockchain to facilitate the aforementioned requirements.

In the healthcare sector, blockchain has the potential to transform the way data is managed, shared, and secured. Drug traceability is a crucial aspect that blockchain technology has the potential to address as it could be used to develop a transparent and traceable supply chain that could mitigate issues pertaining to drug counterfeit and streamline the activities of the supply chain from the supplier to manufacturer and finally the distributors in the supply chain. Furthermore, in the context of clinical trials, blockchain could be used to address challenges related to data integrity and transparency. By recording clinical trial data in a tamper-resistant manner on a blockchain, the stakeholders will be able to access authentic and reliable information pertaining to clinical trials. In the domain of telemedicine, blockchain can be utilized to ensure authentic marketing practices and transparency in telemedicine transactions.

In particular, blockchain can be utilized to create a robust distributed model to securely store patient data and medical records. Since the data is stored in a distributed manner, a mechanism also needs to be established for patients to provide consent on the usage of data by different healthcare providers. Development of such an architecture could be facilitated by blockchain and this study strives to study the possibility of proposing such a novel architecture utilizing blockchain technology.

1.3 - Problem in Brief

Storage of patient data in a secure manner has become a paramount concern in the healthcare industry. Patient data were recorded using traditional file system methods in the early days. With the advancement of technology, this technique was dropped eventually as health records were recorded in a digital format. However, storing data in a digital format in a centralized manner has its fair share of drawbacks.

One major concern is the risk of data breaches and unauthorized access. Centralized storage systems are susceptible to data breaches that can compromise the confidentiality and privacy of sensitive patient information. Furthermore, dependence on a centralized system makes it an attractive target for malicious authors to exploit vulnerabilities and perform malicious practices. As the data is also managed by a central authority such as a healthcare institution, the patients do not have direct control over their medical data.

Blockchain technology offers a promising solution to address the inefficiencies in patient data storage and consent management. To achieve this, a decentralized data storage approach that is facilitated by the principles of blockchain technology can be implemented. By leveraging blockchain, patient data can be securely distributed across a network of nodes, eliminating the vulnerability associated with centralized storage. Each block in the blockchain contains a timestamped and encrypted set of patient records, creating a tamper-resistant and transparent system.

1.4 - Aim & Objectives

1.4.1 - Aim

The research aims to evaluate the feasibility of the application of blockchain technology in the healthcare sector to enhance the secure storage and efficient management of patient data & to develop a novel approach to securely store patient data using blockchain technology which also provides the ability for patient data consent management.

1.4.2 - Objectives

- Conduct a literature review of the existing techniques for patient data storage and consent management.
- Identify and assess inefficiencies within the current methods.
- Investigate the feasibility of implementing blockchain technology for the secure storage of patient data.
- Propose a novel method to securely store patient data using blockchain technology which also provides the ability for patient data consent management.
- To study opportunities, challenges and implications of utilizing blockchain technology in the management of patient data consent management.

1.5 - Proposed Solution

Motivated by the possibilities and probable prospects of the application of blockchain technology in the healthcare sector and to address the existing problems prevalent in the healthcare industry we propose a novel model developed upon the blockchain technology that could be used to effectively manage and securely store patient data which also streamlines the patient data consent management process. Furthermore, the novel model also facilitates anonymous data provision for research and business analytics. To accomplish this task in developing this novel model, the system will be developed and integrated into the following four modules.

- **Module 01: Develop a Blockchain Model to Securely Store Patient Data Develop a Blockchain Model to Securely Store Medical Data and Propose a Novel Algorithm (Cross-Hash Validator Algorithm) to enhance the security of the stored data.**

  This module will provide the foundational architecture for the novel blockchain model. It handles the data structures and the architectural complexities of the system. The following highlights the main aspects that will be covered in this module

  - **The Architecture of the Blockchain platform.**
    - Establishing architecture for the blockchain platform using Java and Spring Boot.
    - Overview of models, controllers, services, and utility classes developed based on the MVC architecture.

  - **Cross-Hash Validator Algorithm**
    - Introduction of a novel algorithm to enhance data security and integrity.
    - Hash Function:

        - Implementation of the SHA-256 hashing algorithm.
        - Generation of unique and secure block data representations.
    - Validation Function:
        - Vertical Validation:
            - Sequential integrity checks of blockchain blocks.
            - Verification of previous hashes to ensure data integrity.
    - Horizontal Validation:
            - Cross-referencing blockchain entries with current records.
            - Ensuring stored data matches actual data to prevent tampering.
- **Algorithm Discussion**
    - Technical aspects and methodologies used in the design and implementation of the algorithm.
    - Evaluation of benefits and drawbacks of the implemented algorithm.
- **UI Design & Development**
    - Focus on creating a user-friendly and intuitive interface for the blockchain platform.

- **Module 02: Enhancing Security, Scalability and Reliability of the Blockchain Model by Creating a Custom Blockchain Network and Consensus Algorithm.**

  Module 02 builds upon Module 01 by providing the ability for the blockchain model to be distributed among multiple nodes (set up the custom blockchain network). It also addresses scalability, security by creating the custom consensus algorithm.
  Module 03 and module 04 are built upon the established principles in modules 01 and 02. Simultaneously,
- **Module 03: Streamlining Patient Data Consent Management Processes**

  Module 03 focuses on optimizing patient data consent management processes, ensuring streamlined and secure procedures.

- **Module 04: Facilitating Anonymous Data Provision for Research and Business Analytics**
  Module 04 is dedicated to facilitating anonymous data provision for research and business analytics, leveraging the robust groundwork laid out by the preceding modules.

Together, these four modules work in synergy to establish an innovative blockchain model to address the problems seen in the healthcare industry today.

1.6 - Summary

This chapter gave an introduction to blockchain technology and its applications in today's world. Furthermore, it also emphasized the limited research conducted to study the feasibility of applying blockchain technology in the healthcare industry and also discussed the

background and motivations behind the research. The chapter also highlighted the current challenges and implications encountered in the healthcare industry today and emphasized how blockchain technology can be used to address these issues. The aim and objectives of the research were also mentioned and finally, the chapter concludes by providing a brief overview of the proposed system after introducing blockchain technology and its applications in contemporary settings.

CHAPTER 02 : LITERATURE REVIEW

2.1 - Introduction

The healthcare industry has adopted blockchain technology as a solution to the major problems with data security, transparency, and interoperability. It is imperative to tackle these concerns to protect patient confidentiality, enhance the effectiveness of care delivery, and minimize administrative workloads. Strong data security is facilitated by the intrinsic qualities of blockchain, such as decentralization and cryptographic principles, which produce an unchangeable, visible, and impenetrable ledger for healthcare data. Additionally, blockchain acts as a single platform for standardized health records, facilitating smooth data sharing and interoperability across various healthcare providers.

One of the major issues facing contemporary healthcare systems is the high expense of oversight and maintenance [1]. With several areas that each include doctors, researchers, practitioners, support personnel, managers, and patients, the healthcare system is extremely complicated. The classification and administration of patient data thus become extremely difficult tasks. The difficulty is compounded by the fact that many healthcare areas have diverse data formats and operations. These factors make it extremely difficult for different healthcare areas to exchange information on healthcare effectively [2]. The development, management, and upkeep of a system is necessary to address health information records and interchange. Traditional personal health record and electronic health record systems are developed and maintained by a third party; trust, privacy, and data security are still significant obstacles. However, stakeholders' demands for privacy cannot be met by the third-party-based healthcare recording systems now in place [3]. Because of privacy and data security concerns, the conventional electronic healthcare approach lacks transparency.

Blockchain technology has tremendous potential in addressing security-related issues and the challenging challenge of massive and extremely diversified data in healthcare systems. Blockchain is a peer-to-peer network, digital ledger, and decentralized, distributed database. The blockchain facilitates the safe transfer of information between parties by connecting several computers through nodes and doesn't require any transactions to create new blocks. The client can use a blockchain that is encrypted to access all authorized and verified medical information. Anybody may add a new chain to the block and select transactions. A blockchain's master key is a hash, and it may provide unique IDs for cryptocurrencies to add data by utilizing this hash function.

Blockchain technology integration has become a viable way to address the ongoing problems that traditional electronic health record (EHR) and personal health record (PHR) systems are facing. These systems' privacy and security flaws have driven up healthcare expenses, which is a hardship on patients as well as healthcare providers. Aware of the shortcomings in the present healthcare infrastructure, interested parties are looking to blockchain technology to create decentralized architectures for the interchange of electronic health information and transform healthcare management systems [4]. Leading healthcare firms, such as IBM, predict that by 2022, blockchain will significantly alter the healthcare sector, with a market expected to grow to over $500 million [5]. This technology change is perceived as a way to improve uniformity, reduce supply chain management inefficiencies, and make use of smart healthcare systems.

Four main areas—conceptual evolution, performance enhancement, technology advancement, and data management—are evident in the literature and are responsible for much of the research being done on blockchain-based healthcare systems.

2.1.1 - Evolution of Concept

The development and improvement of blockchain-based healthcare ideas have been the subject of research. During idea development, this includes the development of algorithms, evidence, and secure communication methods to address data security issues. The focus on frameworks that help create more effective blockchain architectures, like game methods and calculation homomorphism, is noteworthy. Blockchain has applications in the healthcare industry that range from consensus methods like identity verification to proof-of-work systems. The research highlights the need to improve frameworks for effective blockchain integration.

2.1.2 - Advances in Technology

The healthcare industry has witnessed a notable advancement in blockchain technology, namely in the areas of smart ecosystem development, blockchain architectural enhancements, and predictive capacity integration.

2.1.3 - Development of Smart Ecosystems for Healthcare

To enable the development of intelligent healthcare systems, several academics have looked at integrating blockchain technology with healthcare ecosystems. Adoption of blockchain technology has been associated with improving telehealth ecosystems and providing solutions for telecommunications and blockchain-based telehealth.

2.1.4 - Technological Improvements in Blockchain Architecture

Research has focused on improving system performance by tackling issues such as transaction latency, restricted data block geometries, and unknown key exploiters. Problems with memory use, memory load, overheating, and reliable node identification are also addressed.

2.1.5 - Building Full Power of Prophecy

Blockchain technology is developing and integrating AI with healthcare. Researchers are looking at how blockchain can be integrated with big data, networks, photovoltaic cells, cloud computing, edge computing, and the Internet of Things. The goal of these connections is to improve diagnostics and medical information technology by strengthening predictive capacities.

2.1.6 - 2.1.3 Increase Efficiency

Numerous research, with an emphasis on process and technique improvements, have looked into methods to use blockchain technology to improve the efficacy of healthcare systems.

2.1.7 - Procedure

Technical features of blockchain health systems have been the focus of research, with topics including load estimations, energy costs, communication overloads, and integration delays. Future reporting systems, runtime, shipment times, and latency are the areas of focus.

2.1.8 - Method

Enhancing data administration, interoperability, and inter-institutional access are strategies to raise the efficacy of blockchain-based healthcare organizations. Scholars endeavour to create cohesive frameworks and enhance the adaptability of deployed blockchain technology.

2.1.9 - Management of Data

In their emphasis on information privacy, protection, and handling, researchers highlight the significance of data management in the healthcare industry.

2.1.10 - Information Privacy

The goal of preserving confidentiality through secure access to medical information and user verification has been pursued. Blockchain-based architectures provide certified, controlled, and user-oriented methods for handling medical data, including personal health records (PHRs).

2.1.11 - Protection of Information

Studies have investigated the use of multiple identities, biometric authentication, and effective authenticity to accomplish the important goals of safeguarding data and preventing unauthorized access.

2.1.12 - Handling of Data

Researchers have realized that maintaining healthcare information requires following guidelines or norms. Focus areas include gathering information in a lawful manner, preventing information theft, avoiding duplicate storage expenses, and keeping information forever. As capabilities increase, attention is being paid to facilitating data transfer and inter-institutional information sharing.

2.2 - Literature Review of Module 01

2.2.1 - Introduction

Blockchain technology has garnered significant attention for its potential to revolutionize various industries, including healthcare. This review focuses on examining the current research on the application of blockchain technology in healthcare, identifying its drawbacks, and exploring its successful applications in other domains. The goal is to provide insights into the opportunities, challenges, and implications of integrating blockchain in healthcare, particularly for securely storing patient data and proposing novel security algorithms like the Cross-Hash Validator Algorithm.

2.2.2 - Flaws and Challenges in Existing Implementations of Blockchain in Healthcare

1. **Data Duplication & Redundancy**
    - **Immutability and Redundancy**: Blockchain's immutability ensures data integrity and security by preventing unauthorized modifications. However, this feature can lead to significant redundancies and duplication of data within the blockchain. For instance, if a patient's information needs updating, a new block must be added to the chain instead of modifying existing records. This results in multiple blocks containing similar or overlapping data, complicating data management and increasing storage requirements within the blockchain system [61][62]

2. **Lack of Query Optimization**
    - **Inefficient Data Access**: Another critical issue is the lack of query optimization when accessing data directly from the blockchain. The decentralized nature of

blockchain technology often leads to time-consuming processes for querying data, as every transaction must be validated and traversed through multiple blocks. This inefficiency can hinder the responsiveness of applications relying on real-time access to patient data, making it challenging to meet the demands of healthcare providers and other stakeholders in the system.[61]

2.2.3 - Successful Applications of Blockchain in Other Domains

1. **Finance and Banking**
    - **Enhanced Security and Efficiency**: Blockchain has been successfully implemented in the finance sector, enhancing security, transparency, and efficiency in transactions. Cryptocurrencies like Bitcoin and Ethereum have demonstrated the robustness of blockchain in handling financial transactions on a global scale. The technology has reduced the need for intermediaries, lowered transaction costs, and increased transaction speed [61][62].
2. **Supply Chain Management**
    - **Improved Traceability and Transparency**: In supply chain management, blockchain has been used to enhance traceability and transparency. For instance, companies like IBM and Walmart have implemented blockchain to track the journey of products from origin to consumer, ensuring authenticity and reducing fraud. This application demonstrates blockchain's capability to handle complex logistics and maintain an immutable record of transactions [61][62].
3. **Intellectual Property and Digital Rights Management**
    - **Protection and Fair Use**: Blockchain has also been applied in managing intellectual property and digital rights. By creating a tamper-proof record of ownership and transactions, blockchain ensures that creators can securely manage their intellectual property and receive fair compensation for their work. This application highlights blockchain's potential to protect sensitive information and ensure fair use and distribution [61][62].

While blockchain technology holds great promise for transforming the healthcare sector, several challenges must be addressed to realize its full potential. The current research highlights the need for scalable, interoperable, and secure blockchain solutions that comply with regulatory standards. Learning from successful implementations in other domains can provide valuable insights for overcoming these challenges. Further research and development, particularly in novel algorithms like the Cross-Hash Validator, are essential to enhance the security and efficiency of blockchain applications in healthcare.

2.2.4 - Conclusion

While blockchain technology holds great promise for transforming the healthcare sector, several challenges must be addressed to realize its full potential. The current research highlights the need for scalable, interoperable, and secure blockchain solutions that comply with regulatory standards. Learning from successful implementations in other domains can provide valuable insights for overcoming these challenges. Further research and development, particularly in novel algorithms like the Cross-Hash Validator, are essential to enhance the security and efficiency of blockchain applications in healthcare.

2.3 - Literature Review of Module 02

2.3.1 - Blockchain Network

According the literature, One of the major issues in existing blockchain networks is the inability of nodes to monitor their neighbors continuously, especially after a node failure and recovery. This often results in discrepancies in the blockchain data across the network.

2.3.2 - Consensus Algorithm

2.3.2.1 Introduction

Consensus algorithms are critical for validating the validity of distributed transactions, maintaining consistency amongst state machine replicas, and confirming the identities of distributed nodes. They have a variety of applications, including cryptocurrencies like Bitcoin, distributed lock services like Google's Chubby (which uses the Paxos algorithm), and real-world challenges like distributed system time synchronisation and network load balancing. [36] Consensus algorithms are also useful for ranking webpages and managing many UAVs, robots, and agents. Blockchain technology, which has gained popularity since 2008, uses these techniques to keep a tamper-proof record of transactional data. [36]

2.3.2.2 Proof of Work (PoW)

PoW is one of the most commonly known and often used consensus algorithms, which came to the surface originally in the Bitcoin. In PoW, miners compete against each other by solving highly complex mathematical problems using the processing power of their machines. The miner who manages to solve the problem first wins the reward of adding a new block to the blockchain and thus receives a prize. This approach is very safe and ensures the absence of a central power but is criticized for a great amount of energy use and too slow that it is not practical for big business.

2.3.2.3 Proof of Stake (PoS)

This algorithm is created to solve the problem of energy inefficiency of PoW. In PoS, decide who will add the next block to the blockchain by various combinations of random selection weighted by the size of a user's stake (i.e., the relative number of coins the user has) in the cryptocurrency". This algorithm is less energy consuming. Also, it allows for quicker block creation. However, In PoS rich one will be richer, that happened often in this algorithm based blockchains. A user with a higher stake to mine blocks more likely, "the richer get richer". PoS often leads to the "nothing at stake" issue, meaning that nodes may support several blockchains without any risk.

2.3.2.4 Delegated Proof of Stake (DPoS)

Delegated Proof of Stake (DPoS) is a improved version of Proof of Stake (PoS). In this algorithm they introduced a voting system. Network participants (nodes) can vote for validating the transactions and add new blocks to the blockchain. DPoS is more efficient and scalable than PoS but it increases the centralization as only a few delegates control the whole blockchain network.

2.3.2.5 Proof of Elapsed Time (PoET)

Proof of Elapsed Time (PoET) algorithm developed by Intel Corporation. In this algorithm, just randomly select a node to add the next block to the blockchain, because they believe it ensures the energy efficiency and fairness. PoET completely relies on trusted hardware. Also, on security of selection process which have some concerns about centralization and dependency on specific hardware.

2.3.2.6 Practical Byzantine Fault Tolerance (PBFT)

Practical Byzantine Fault Tolerance (PBFT) specially designed for handling Byzantine faults.
What are Byzantine faults?
Byzantine fault is nodes may fail to be keep active or nodes act maliciously.
So, in this algorithm nodes have communicated each other and before the add next block they should agree on adding that block. To add next block to the network, required two-thirds of the nodes to reach a consensus This method is efficient and fast also energy saving but this algorithm is not suitable for a network that has large number of nodes, because when increase the nodes of the network, this algorithm is becoming not practical. Also compared to other algorithms, this algorithm is leading to scalability problems.

2.3.2.7 Delegated Byzantine Fault Tolerance (dBFT)

This Delegated Byzantine Fault Tolerance (dBFT) algorithm is similar to PBFT algorithm but in this algorithm uses a voting system to select a small number of nodes to perform consensus tasks in the network. Because of the small number of nodes, this algorithm reduces the communication overhead of PBFT by enhancing the scalability. But this algorithm also introduces a level of centralization, as selected small number of nodes hold significant power in the blockchain network.

2.3.2.8 Proof of Weight (PoWeight)

Proof of Weight (PoWeight) algorithm is completely weight-based algorithm. It assigns weights to user-based factors like account balance, reputation, etc. This algorithm is customizable and scalable but in this algorithm user participation can be a question because they don’t get rewards by participation.

2.3.2.9 Proof of Burn (PoB)

Proof of Burn (PoB) is an algorithm that requires user to spend/burn or destroy a portion of their cryptos to gain mining privileges. This algorithm is energy efficient because this doesn’t rely on computation power. But this method leads to a short-term loss for a potential long-term gain. Also, may lead to centralization as wealthier participants can afford to burn more cryptos.

2.3.2.10 Proof of Capacity (PoC)

Proof of Capacity (PoC) algorithm also known as Proof of Space. This algorithm uses free hard drive spaces for mining mechanism. Miners can plot their hard drives with data and each one is compared with a network’s puzzle and one who has the closest match hard drive wins the block. This algorithm is energy efficient and accessible for everyone as it only requires regular hard drives. But this also leads to centralization, if there are participants who can allocate more storage space.

2.3.2.11 Proof of Importance (PoI)

Proof of Importance (PoI) algorithm was introduced by NEM. This algorithm specially developed to address the flaws in PoS algorithm. PoI assigns scores to each account based on its balance and activities. This motivates participants to both holding and spend coins. This algorithm is energy efficient. And also resistant for most of the attacks like Sybil attacks.

2.3.2.12 Proof of Activity (PoA)

Proof of Activity (PoA) algorithm is a combination of PoW and PoS. In this algorithm miners use PoW to mine an empty block, and with the help of PoS, we can choose a group of validators. So, they can add new blocks to the blockchain. This algorithm increases security and decentralization but has the energy inefficiency because of the PoW.

2.3.2.13 Directed Acyclic Graphs (DAGs)

Directed Acyclic Graphs (DAGs) algorithm is completely based on data structures. This used in some cryptocurrencies instead of traditional blockchains. In DAGs, every transaction is linked with other transactions rather than grouped into blocks. It allows to faster validation and lower transaction fees. DAGs are scalable and decentralized. Therefore, this algorithm suitable for high frequency transactions, such as transactions in IoT networks.

2.4 - Literature Review of Module 03

The healthcare industry has been facing significant challenges in managing patient consent and data privacy, particularly with the advent of new technologies and the increasing demand for personalized, data-driven healthcare solutions. Blockchain technology, characterized by its decentralization, transparency, and immutability, offers a promising solution to these issues. This literature review explores the current state of research on applying smart contracts for consent management in healthcare, highlighting the benefits, challenges, and potential future developments.

Blockchain's decentralized nature ensures that no single entity controls the data, enhancing security and trust across the system [1, 6]. The inherent transparency and immutability of blockchain are crucial, as they allow for verifiable and tamper-proof records, which are essential for maintaining the integrity of patient data [3, 5]. Secure data storage and record-keeping are achieved by storing patient data off-chain using decentralized storage solutions like IPFS, while the blockchain records transaction histories and access logs [7, 10]. Furthermore, the use of advanced cryptographic techniques ensures the security and privacy of medical documents and personal data [3, 9].

Smart contracts, a fundamental component of blockchain technology, enable dynamic consent management, allowing patients to efficiently grant, modify, and revoke consent [2, 4, 7]. By automating complex consent processes through self-executing code, smart contracts ensure compliance with data protection regulations such as the General Data Protection Regulation (GDPR) [1, 5, 6]. Fine-grained access control mechanisms are also facilitated by smart contracts, enabling patients to specify the purposes for which their data can be accessed, thus enhancing patient control over data usage [7, 8].

Interoperability with existing technologies and frameworks is another significant advantage of blockchain-based consent management systems. Roman-Martinez et al. proposed blockchain-based service-oriented architecture (SOA) for consent management, access control, and

auditing in the healthcare domain enables tamper-proof and immutable storage of subject of care (SoC) consents on a blockchain, providing fine-grained access control services to protect health data according to SoC consents [5]. Roman-Martinez et al. introduced the adoption of healthcare standards such as FHIR and XACML ensures seamless integration and interoperability among different stakeholders in the healthcare ecosystem [5, 6]. Additionally, existing frameworks like the Ocean Protocol and Hyperledger Fabric are leveraged to develop robust consent management solutions [2, 6, 7]. These systems are designed to comply with data protection regulations, particularly the GDPR, emphasizing user-centric consent and privacy by design principles [1, 6]. They also provide comprehensive audit trails and accountability features, supporting regulatory compliance and oversight [1, 5, 6].

Innovative approaches to consent management in healthcare have been proposed in the literature. Al Amin et al. proposed blockchain-based systems for managing consent in clinical studies offer transparency, efficiency, and ease of verification and change management [2, 8]. These systems focus on ethical AI application development in healthcare by ensuring secure and standardized consent processes [2, 8]. Additionally, some architectures support emergency access permissions, where patients can authorize trusted entities to make access decisions on their behalf [3, 10]. These solutions integrate reputation-governed trusted oracles and threshold cryptography to facilitate secure decision-making in emergency scenarios [3, 10].

National and large-scale implementations of blockchain-based consent management frameworks have also been explored by Sharma et al [9]. For instance, frameworks proposed for national healthcare schemes address challenges related to data interoperability, privacy, and fraud prevention on a large scale [9, 10]. According to Amofa et al. These implementations emphasize the secure sharing of personal health data in health information exchanges, ensuring that only authorized users can access and perform permitted operations on the data through the use of cryptographic keys and smart contracts [9, 10].

Further context is provided by emphasizing the evolving technologies and increasing demand for personalized, data-driven healthcare solutions as the primary motivators for better consent management and data privacy in healthcare. The potential of smart contracts to give individuals greater control over their personal data and the ability to grant or revoke access as needed is highlighted.

Merlec et al. detailed benefits and specific applications of blockchain, such as GDPR compliance through user-centric control over personal data collection and consent usage [1], addressing the limitations of traditional paper-based Written Informed Consent (WIC) processes [2], and utilizing frameworks like the Ocean Protocol for decentralized and secure consent management by Jung and Pfister [2]. Madine et al proposed the implementation of decentralized IPFS storage, reputation-governed trusted oracles, and threshold cryptography in fully decentralized multi-party consent management solutions [3] is elaborated upon, alongside the detailed architecture and evaluation of such systems [3].

Technical details and implementation insights include a Smart Contract-based Dynamic Consent Management System (SC-DCMS) with components like decentralized IPFS storage nodes and blockchain transaction history [1]. The implementation of this prototype system on the Quorum blockchain platform is assessed for acceptability, reliability, security, privacy, and performance [1]. Additionally, the architecture of a blockchain-based consent management system designed to meet GDPR requirements, including a permissioned blockchain (Hyperledger Fabric), a REST API, and an external database, is explored by Aldred et al. [6]. The system uses hashed user-company pairs to represent permissions, allowing users to control

access without being directly traceable [6], and is designed to meet privacy by design principles, ensuring proactive privacy protection, end-to-end security, and transparency [6].

Agbo and Mahmoud introduced comprehensive blockchain-based e-Health consent management framework for managing consent for data access and processing utilizes Hyperledger Fabric for security and privacy [7]. This framework includes the similar features that Al Amin et al. introduced like patient-provider agreement, consent generation, modification, withdrawal, expiration, and archiving components in a blockchain-based system for managing informed consent for clinical diagnosis and treatment [8]. Utilizing Hyperledger Fabric permissioned blockchain ensures security and privacy, with peer nodes storing identical copies of PMRs and smart contracts controlling data access. Patients can grant or revoke access permissions, with the system designed to meet the core objective of achieving effective consent management in PMR processing [7].

In conclusion, this review underscores the significant potential of blockchain and smart contract technologies to revolutionize patient consent management in healthcare. These technologies offer enhanced security, transparency, and patient control over personal data while addressing regulatory compliance and interoperability challenges. Future developments are likely to focus on refining these technologies for broader adoption and integration into existing healthcare systems.

2.5 - Literature Review of Module 04

In the field of secure data provision, the convergence of blockchain technology and advanced cryptographic techniques like Attribute-Based Encryption (ABE) has gained significant attention. This literature review aims to contextualize the proposed integration of ABE with a Java-based blockchain by examining existing works and highlighting their contributions, limitations, and areas for further research.

2.5.1 - Blockchain Technology in Data Security

Blockchain technology is widely recognized for its capabilities in providing decentralized, immutable, and transparent data storage. Nakamoto's seminal work on Bitcoin (2008) introduced the concept of blockchain as a secure ledger for cryptocurrency transactions. Since then, blockchain's applicability has expanded to various domains, including healthcare, supply chain management, and finance

2.5.1.1 Key Features:

Decentralization: Eliminates the need for a central authority, reducing the risk of single points of failure.
Immutability: Ensures that once data is written, it cannot be altered, thus maintaining data integrity.
Transparency: All transactions are recorded on a public ledger, enabling auditability and traceability.

2.5.1.2 Limitations:

Scalability Issues: Traditional blockchain systems often struggle with scalability, as seen in Bitcoin and Ethereum, where transaction throughput is limited.
Energy Consumption: Proof of Work (PoW) consensus mechanisms are energy-intensive and not environmentally sustainable.

2.5.2 - Attribute-Based Encryption (ABE)

ABE is an advanced cryptographic technique that enhances data security by allowing encryption and decryption based on attributes rather than individual keys. Sahai and Waters (2005) introduced ABE, which supports fine-grained access control by associating access policies with ciphertexts.

2.5.2.1 Key Features:

Fine-Grained Access Control: Enables specific access policies, ensuring that only users with the required attributes can decrypt the data.
Scalability: ABE is scalable for environments with many users and varying access control requirements.
Privacy Preservation: Maintains data confidentiality by ensuring that only authorized users can access sensitive information.

2.5.2.2 Limitations:

Complex Key Management: The management of attributes and keys can become complex in large systems.
Performance Overhead: The encryption and decryption processes in ABE can be computationally intensive.

2.5.3 - Integrating ABE with Blockchain

The integration of ABE with blockchain technology aims to leverage the strengths of both to provide a secure, scalable, and privacy-preserving data sharing solution. Several studies have explored this integration, focusing on different aspects of data security and access control.

2.5.4 - ABE-Based Blockchain Systems:

Healthcare Data Sharing:
Research: Li et al. (2018) proposed a blockchain-based system for secure sharing of healthcare data using ABE. Their system ensures that only authorized healthcare providers can access patient records.
Contributions: Enhanced data privacy and security in healthcare environments.
Limitations: High computational overhead due to complex ABE operations.

IoT Security:
Research: Zhang et al. (2019) developed an IoT security framework using blockchain and CP-ABE (Ciphertext-Policy ABE). The framework secures data transmission in IoT networks.
Contributions: Improved data security and access control in IoT systems.
Limitations: Scalability issues due to the large number of IoT devices.

Supply Chain Management:
Research: Feng et al. (2020) integrated ABE with blockchain for secure data sharing in supply chain management. The system ensures that only authorized entities can access sensitive supply chain data.
Contributions: Enhanced traceability and security in supply chain operations.
Limitations: Performance overhead and complexity in managing access policies.
Comparison with Java-Based Blockchain Systems:
While the aforementioned studies highlight the potential of ABE-based blockchain systems, there is limited research focusing specifically on Java-based blockchain implementations. Java,

as a programming language, offers robustness, portability, and a wide range of libraries that can facilitate the development of blockchain systems.

2.5.5 - Proposed Approach:

The proposed system integrates ABE with a Java-based blockchain to securely store and manage medical data. This approach addresses the limitations identified in previous studies by leveraging Java's capabilities to enhance system performance and manageability.

2.5.6 - Key Contributions:

Robust Implementation: Java-based implementation ensures robustness and portability across different platforms.
Efficient Key Management: Improved management of attributes and keys using Java's extensive libraries.
Scalable Solution: Enhanced scalability by optimizing the integration of ABE with blockchain, reducing computational overhead.

2.5.7 - Novelty and Advantages:

Dynamic Access Control: The integration allows for dynamic and fine-grained access control, ensuring that only authorized users with specific attributes can access sensitive medical data.
Enhanced Privacy and Security: By storing encrypted data on a decentralized and immutable blockchain, the system ensures data integrity and privacy.
User-Friendly Data Provision: The use of query links for data access simplifies the process for authorized users, making the system more user-friendly.

2.6 - Summary

A thorough examination of a wide range of topics, such as conceptual development, technology breakthroughs, productivity gains, and data management, is shown by the body of material now available on blockchain integration in the healthcare industry. When it comes to solving the problems that beset contemporary healthcare systems, researchers have made tremendous progress. In order to further blockchain technology's potential for revolutionary change in the healthcare industry, the study does acknowledge the necessity for more research in this area.

CHAPTER 03 : TECHNOLOGY ADAPTED

3.1 - Introduction

This chapter discusses the technology adapted and resources utilized in conducting the research to study the feasibility of applying blockchain technology in the healthcare sector. It explores the programming languages used in developing the blockchain model, the development environment and other tools utilized, and the software utilized for version control.

3.2 - Technologies Adapted

The following technologies were adopted and utilized in conducting the research on the feasibility of applying blockchain technology in the healthcare sector.

3.2.1 - Programming Languages

3.2.1.1 Java

One of the main objectives of conducting this research is to propose a novel method using blockchain technology in order to securely store patient data. To facilitate this task, it was necessary for our team to develop a blockchain model. Java was primarily used in defining the underlying architecture of the blockchain model and It was also used to define the necessary data structures, classes and methods.

Figure 1: Java Logo

3.2.1.2 Solidity

Another quintessential aspect of every blockchain model is smart contracts. A smart contract is a piece of code that enforces certain rules in the blockchain model. Solidity is a programming language that was designed specifically for writing smart contracts in order to facilitate decentralized blockchain networks.

Figure 2: Solidity Logo

3.2.2 - Development Environment/Tools

3.2.2.1 IntelliJ Idea

IntelliJ idea was the integrated development environment which was utilized to set up the required Java environment and to develop the architecture of the blockchain model.

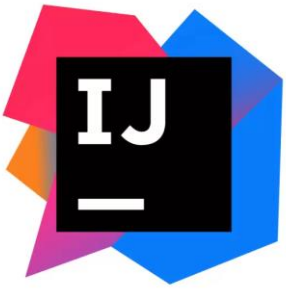


Figure 3: IntelliJ Logo

3.2.2.2 Postman

Postman is a collaboration platform for API development and testing. In this study, Postman was utilized to test the developed APIs.

Figure 4: Postman Logo

3.2.3 - Version Controlling

Git and GitHub were primarily used for version-controlling purposes. It helped the team to collaborate and manage source code in an efficient manner.

Figure 5: GitHub Logo

3.3 - Summary

This chapter discussed the technology adopted in our study of assessing the feasibility of utilizing blockchain technology in the healthcare sector. It also highlighted the programming languages and other development tools that are utilized in developing the novel model.

CHAPTER 04 : APPROACH

4.1 - Introduction

The approach is summarized in this chapter, which is separated into four main modules. Also, this describes how each individual research module fits into the overall solution, including desired users, processes, inputs, and outputs. Furthermore, this chapter examines the proposed solution's design in terms of how it would be implemented utilising the technologies discussed in Chapter 3.
This research explores the integration of blockchain technology into the healthcare sector, with each module spearheaded by a specific member, introducing unique contributions to the overall framework. The objective is to create a secure and reliable blockchain model tailored for healthcare applications, addressing existing challenges related to data security, privacy, data classification, frauds of drugs, scalability and how to share the health data to business analytics or research without leaking sensitive data.

4.2 - Module 01: Develop a Blockchain Model to Securely Store Medical Data and Propose a Novel Algorithm (Cross-Hash Validator Algorithm) to Enhance the Security of the Stored Data

Input

The primary inputs for developing a blockchain model to securely store medical data encompass various components. This includes medical data, that will be stored in the blockchain network, security protocols, hashing algorithms and healthcare laws and regulations for managing medical data in digital format.

Process

This development of the blockchain model is considered here. This involves a novel model to facilitate efficient management and secure storage of patient data in a decentralized manner.

Output

Novel and secure blockchain model, designed to meet the specific demands of healthcare data storage. Furthermore, a novel algorithm will be used to hash and validate the entries of the blockchain.

4.3 - Module 02: Enhancing Security, Scalability and Reliability of the Blockchain Model by Creating a Custom Blockchain Network and Consensus Algorithm.

Input

The inputs for the enhancing security, scalability and reliability of the blockchain model is primarily the blockchain with the data, consensus algorithms and distributing network protocols.

Process

Creating a custom consensus algorithm specifically designed for the healthcare industry is crucial to enhance the security and reliability of the blockchain model. This algorithm will be tailored to address the unique requirements and challenges of the healthcare sector, such as the need for privacy, data integrity, and permissioned access.
Ensuring the security and reliability of the blockchain model in the healthcare industry is of utmost importance. Measures will be implemented to prevent unauthorized access, tampering, and fraud, thereby establishing trust and confidence in the blockchain network. Additionally, the algorithm will prioritize fault tolerance and resilience to mitigate the risk of system failures and disruptions. By implementing stringent security and reliability measures, the healthcare industry can maintain the integrity and trustworthiness of its blockchain-based solutions.
Scalability issues will be addressed through the blockchain network and the consensus algorithm, using Springboot framework and java has developed the p2p blockchain network with the gossip protocol. Since, hospitals or medical centres needs limited number of nodes to maintain this network custom gossip protocol can easily address the scalability issues and reliability of the blockchain network. By addressing scalability issues, security and reliability the healthcare industry can ensure the long-term viability and effectiveness of blockchain technology.

Output
A secure, reliable and scalable blockchain model with a custom blockchain network & the custom consensus algorithm, mitigating security concerns and enhancing overall reliability.

4.4 - Module 03: Streamlining Patient Data Consent Management Processes

Input
The primary inputs for this module will be the guidelines needed to be followed when specifying the consent management process with regard to patient data.

Process
Focuses on streamlining patient data consent management processes. This involves advanced identity verification methods, efficient consent initiation, granting and revocation procedures, and the incorporation of privacy-preserving techniques to address existing privacy concerns in healthcare data management.

Output
An ethical consent management system that empowers patients with control over their data while adhering to regulatory frameworks.

4.5 - Module 04: Facilitating Anonymous Data Provision for Research and Business Analytics

Input
The primary inputs for this module will be the unencrypted patient data that needs to be encrypted so that it could facilitate the provision of patient data in an anonymised manner for research and business analysis. The growing need for secure and anonymous data sharing in research and business analytics necessitates advanced cryptographic solutions. This paper explores the use of Attribute-Based Encryption (ABE) within a blockchain framework to provide a secure, decentralized, and anonymous method for data provision.

Process

The effort to facilitate anonymous data provision for research and business analytics.
Data Collection: Sensitive medical data is collected and prepared for encryption.
Encryption Using ABE: The data is encrypted using public keys derived from attributes. This process ensures that only users with specific attributes can decrypt the data, providing fine-grained access control. Public keys are generated and managed securely.
The encrypted data is stored in a Java-based blockchain, ensuring immutability and security. Each block contains the encrypted data and metadata, including timestamps and previous block hashes. Blockchain operations are managed to ensure the integrity and security of the stored data.

Output

Decryption Key Issuance: Authorized external parties are issued decryption keys based on their attributes. These keys are generated securely and provided to authorized users.
Query Link Creation: A query link is generated for accessing the decrypted data. This link includes necessary information for accessing the blockchain and decrypting the data. Authorized users can use their keys to access and decrypt the data through the provided link.

4.6 - Summary

This chapter outlines the comprehensive approach adopted for integrating blockchain technology into the healthcare sector. Each module plays a pivotal role in constructing a robust and ethical framework for handling patient data securely and transparently.

CHAPTER 05 : ANALYSIS & DESIGN

5.1 - Introduction

This chapter discusses the high-level design and the architecture of the proposed blockchain model that will be used to efficiently manage and securely store patient data. Furthermore, subsequent sections of this chapter dive deeper into the aspects of each module while exploring the mechanism by which each module contributes to the overall outcome of the proposed novel model.

5.2 - High-Level Architecture of the overall system

The underlying architecture of the proposed novel model of blockchain is built upon the groundwork laid out by each of the 04 modules mentioned below.

- **Module 01: Develop a Blockchain Model to Securely Store Medical Data and Propose a Novel Algorithm (Cross-Hash Validator Algorithm) to Enhance the Security of the Stored Data**

  Module 01 lays the foundational architecture for the novel blockchain model. It handles the data structures and the architectural complexities of the system. This module also proposes a novel algorithm called Cross-Hash Validator Algorithm that provides a unique way of handling and storing user data.

- **Module 02: Enhancing Security, Scalability and Reliability of the Blockchain Model**

  Module 02 builds upon Module 01 by providing the ability for the blockchain model to be distributed among multiple nodes. It also addresses scalability, security and other related issues.
  Module 03 and module 04 are built upon the established principles in modules 01 and 02. Simultaneously,
- **Module 03: Streamlining Patient Data Consent Management Processes**

  Module 03 focuses on optimizing patient data consent management processes, ensuring streamlined and secure procedures.

- **Module 04: Facilitating Anonymous Data Provision for Research and Business Analytics**
  Module 04 is dedicated to facilitating anonymous data provision for research and business analytics, leveraging the robust groundwork laid out by the preceding modules.

Collectively, these four modules harmonize to construct the novel blockchain architecture which could be used to securely store patient data while enhancing security and scalability, by streamlining the process of patient data consent management and facilitating anonymous data provision for analytical purposes.

The following diagram depicts the high-level architecture of the blockchain model

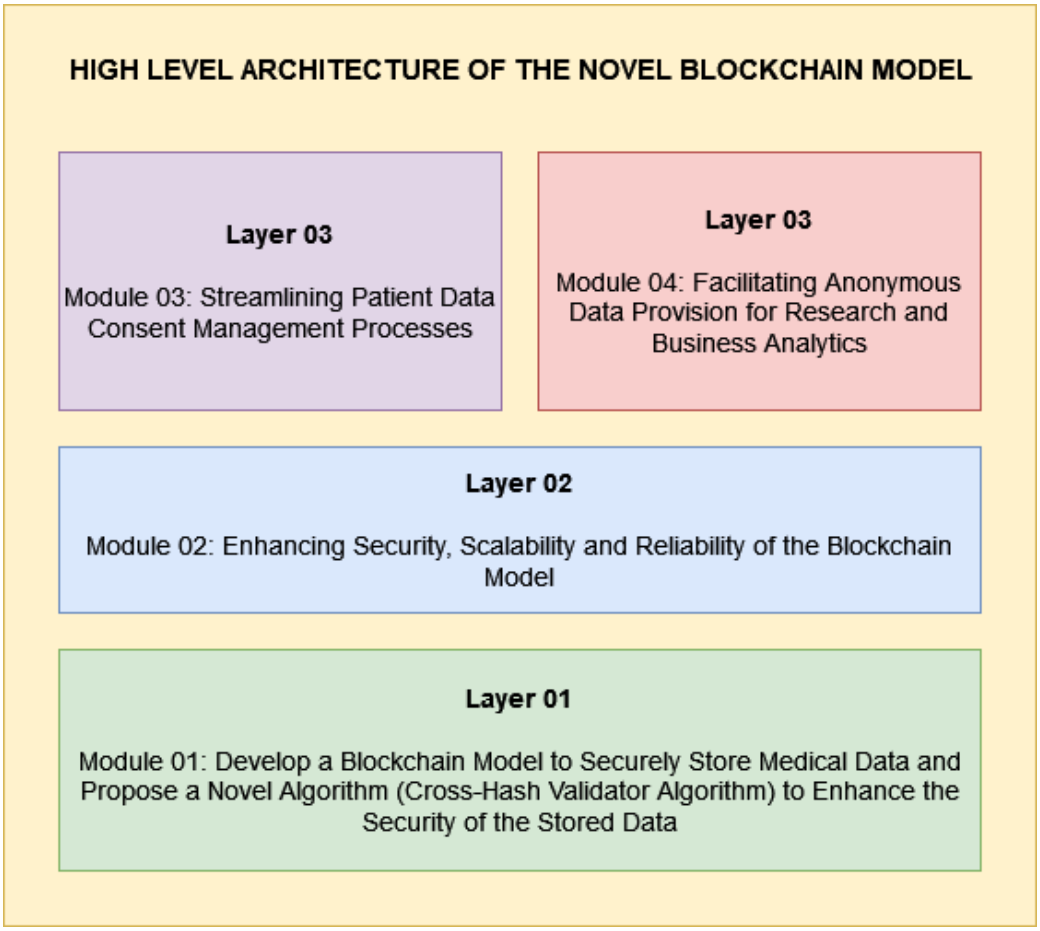


Figure 6: High-level architecture of the novel blockchain model

5.3 - High-Level Architecture of the Individual Modules

5.3.1 - Module 01: Develop a Blockchain Model to Securely Store Medical Data and Propose a Novel Algorithm (Cross-Hash Validator Algorithm) to enhance the security of the stored data.

5.3.1.1 Introduction

Module 01 will lay the foundation for the entire research, with its primary focus on establishing a firm architecture using technologies such as Java and Spring Boot to store medical data with blockchain technology. This module will introduce the basic concepts and initial setup required to build a secure and efficient data storage system.

This module will also look into the detailed architecture of the blockchain platform which was specifically designed for securely storing medical data. It will cover the data structures, and security mechanisms that ensure the efficient and secure storage and retrieval of patient information. The blockchain model was designed upon the MVC(Model-View-Controller) architecture. Therefore, various models, controllers, services and other utility classes that were developed will be further explored in this section.

Furthermore, this module will also discuss a novel algorithm (Cross-Hash Validator Algorithm) designed, developed and implemented to the core of this blockchain platform which would inevitably enhance the security and integrity of the stored patient data. It employs a hashing mechanism and two primary validation mechanisms: vertical validation and horizontal validation.

- Hash Function:
    - The hash(String input) method was developed using the famous "SHA-256 hashing algorithm". This custom hash function that was developed is essential for generating a unique and secure representation of block data and records that will be stored in data stores, which are used in the validation processes.
    - The hashing method incorporates both vertical and horizontal hashing concepts, ensuring comprehensive validation of the data.
        - **Vertical Hashing**: This involves generating hashes that link each block to its predecessor in the blockchain. It ensures the structural integrity of the blockchain by creating a chain of hashes where each block's hash is based on its data and the hash of the previous block.
        - **Horizontal Hashing**: This involves generating hashes for the individual records stored in the database. It ensures the integrity of the actual data by creating unique hashes for each database transaction (POST, PUT, DELETE), which are then stored in the blockchain.
    -
- Validation Function
    - The Validation Function performs two forms of validation.
        - Vertical Validation:
            - This part of the algorithm sequentially validates the integrity of each block in the blockchain.

- checks that each block's previous hash matches the actual hash of the preceding block.
- If any block's previous hash does not align with the hash of the previous block, it indicates a compromise in data integrity.

- Horizontal Validation:
  - This mechanism cross-references the current state of the records with the blockchain entries.
  - It checks that the data stored in the blockchain matches the actual data, ensuring no tampering has occurred.

The design and implementation of the said algorithm will be further discussed in the report, detailing the technical aspects and methodologies used.
Furthermore, the benefits and drawbacks of the newly developed and implemented algorithm, offer a thorough evaluation of its implementation.
The final part of this module focuses on UI design & development, which is crucial for ensuring a user-friendly and intuitive experience whilst using the blockchain platform.
Here are some highlights of the aspects that will be discussed under Module 01.

- **The Architecture of the Blockchain platform.**
  - Establishing architecture for the blockchain platform using Java and Spring Boot.
  - Overview of models, controllers, services, and utility classes developed based on the MVC architecture.

- **Cross-Hash Validator Algorithm**
  - Introduction of a novel algorithm to enhance data security and integrity.
  - Hash Function:
    - Implementation of the SHA-256 hashing algorithm.
    - Generation of unique and secure block data representations.
  - Validation Function:
    - Vertical Validation:
      - Sequential integrity checks of blockchain blocks.
      - Verification of previous hashes to ensure data integrity.
  - Horizontal Validation:
      - Cross-referencing blockchain entries with current records.
      - Ensuring stored data matches actual data to prevent tampering.
- **Algorithm Discussion**
  - Technical aspects and methodologies used in the design and implementation of the algorithm.
  - Evaluation of benefits and drawbacks of the implemented algorithm.
- **UI Design & Development**
  - Focus on creating a user-friendly and intuitive interface for the blockchain platform.

5.3.1.2 The intuition behind blockchain technology

Blockchain technology has revolutionized the way we approach data security, transparency, and trust across various industries. By decentralizing data storage and ensuring immutability, blockchain offers a robust solution for securely recording transactions and managing information.

Developing a custom blockchain platform tailored to cater in storing patient data effectively and efficiently was a challenging task. However, by doing so it gave us the flexibility in coming up with our own architecture on how we define the data structure of block that build up a blockchain.

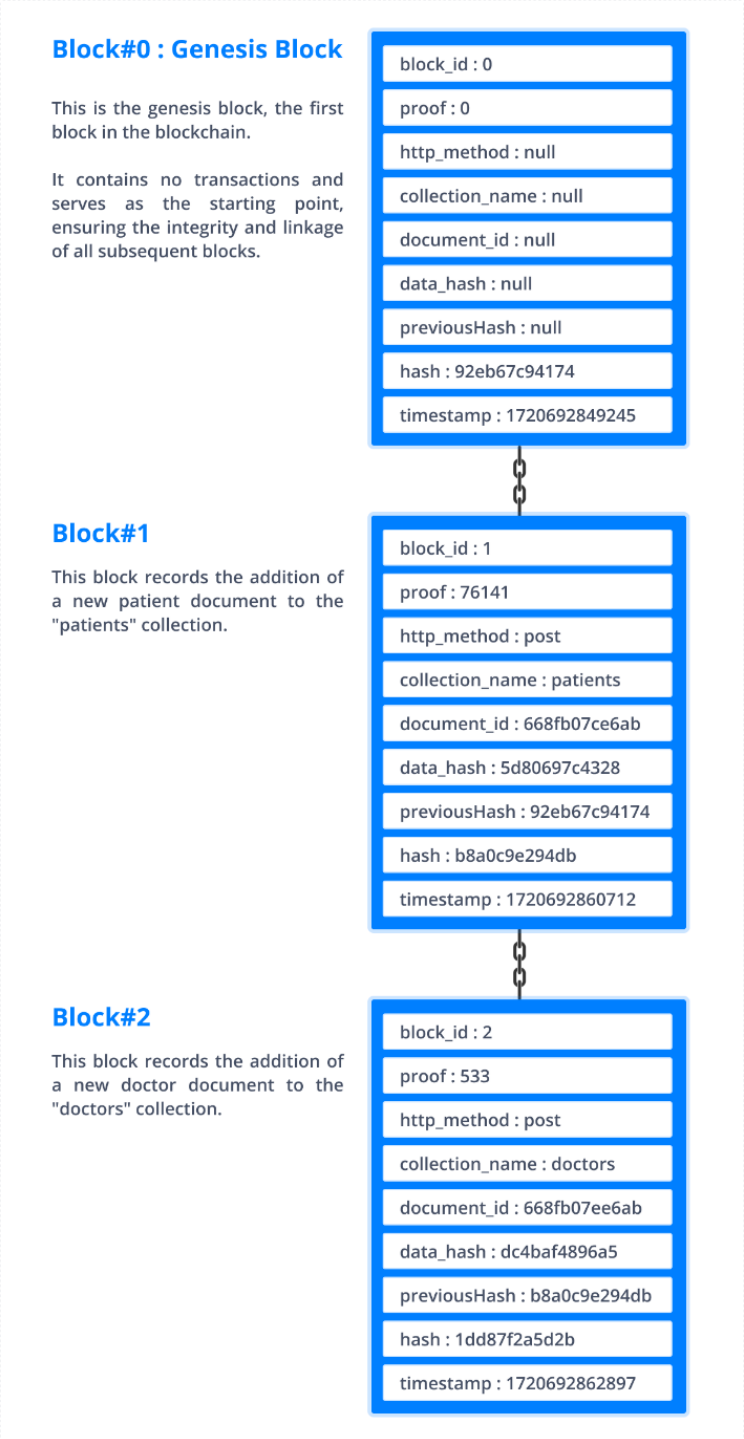


Figure 7: The underlying data structure of the blocks in the blockchain

The above diagram illustrates the basic architecture that governs a blockchain. It also defines the data structure of a block which is the foundation of any blockchain platform.
In our custom blockchain platform developed specifically for storing medical data, I came up with a data structure specifically designed for the use case of storing medical data in a secure manner. All the fields found in a typical block of a blockchain is in detailed below.

- block_id
    - The block_id is a unique identifier for each block in the blockchain. It indicates the position of the block in the chain, starting from 0 for the genesis block.

- proof
    - The proof field is the value that the blockchain miners aim to figure out during the mining process.
    - The inclusion of the proof field gives each block in the blockchain flexibility in deriving the unique hash value for each block.
    - It is a number that satisfies certain conditions defined by the blockchain's consensus algorithm. In our custom blockchain, we came up with a custom consensus algorithm that is developed specifically the use case of storing medical data efficiently.
- http_method
    - This field indicates the HTTP method used to perform an operation on a specific record in the MongoDB database. It corresponds to actions such as creating, updating, or deleting a record in the relevant collection. For example, when a new patient is created, the HTTP method would be POST, while updating that patient's information would use PUT, and removing the patient record would use DELETE.
- collection_name
    - This field specifies the name of the MongoDB collection that is affected by the transaction. It helps in categorizing the data changes. In this blockchain platform, collections include patients, doctors, appointments, insurance_agency, lab_tests, prescriptions, etc.
- document_id
    - This is the unique identifier for the document within the MongoDB collection that is affected by the transaction. It ensures that specific data changes can be tracked precisely.
- data_hash
    - This is the cryptographic hash of the data associated with the transaction. It ensures data integrity by providing a unique fingerprint for the data. For every POST, PUT, and DELETE method, a unique hash corresponding to the MongoDB document will be generated and stored in the block.
    - For an example when creating a patient record in the MongoDB database, the data hash is generated based on the patient's information.
    - This process will be further explained in the Cross-Hash Validator algorithm section.

- previousHash
    - This is the hash of the previous block in the chain. It links the current block to the previous one, ensuring the immutability and integrity of the blockchain.
- hash
    - This is the hash of the current block. It is a unique identifier that includes the block's data and the previousHash, ensuring the block's authenticity.
- Timestamp
    - This is the time when the block was created, recorded as a Unix timestamp in milliseconds. It provides a precise record of when the transaction occurred.

The genesis block is the first block in a blockchain, serving as the foundation for all subsequent blocks. It is unique because it has no predecessor, and it initializes the blockchain's structure and parameters, setting the stage for future transactions and data integrity.
In the genesis block, the initial values for each field are set to establish the foundation of the blockchain. Here's how the data is initialized for each field in the genesis block.

- **block_id:** Set to 0 to denote the first block in the chain.
- **proof:** Initialized to 0, representing the starting point for the proof of work or proof of stake mechanism.
- **http_method:** Set to null, as there are no transactions or operations performed in the genesis block.
- **collection_name:** Also set to null, since no specific MongoDB collection is affected at this stage.
- **document_id:** Set to null, as there are no documents created or referenced yet.
- **data_hash:** Initialized to null, indicating no data exists in the blockchain at this point.
- **previousHash**: Set to null, as there is no preceding block to link to.
- **hash:** Contains a unique hash generated for the genesis block itself, ensuring its integrity and identity within the blockchain.
- **timestamp:** Recorded as the current time in milliseconds, marking the creation of the genesis block.

This initialization provides a clear starting point for the blockchain, enabling subsequent blocks to build upon it with meaningful data and transactions.
In summary, This section outlines the architecture and data structure of a custom blockchain platform designed for securely storing medical data. Each block in the blockchain contains specific fields, including **block_id**, **proof**, **http_method**, **collection_name**, **document_id**, **data_hash**, **previousHash**, **hash**, and **timestamp**, each serving a distinct purpose in maintaining the integrity and traceability of transactions. The genesis block, as the foundational block, is initialized with default values to establish the blockchain's structure, providing a clear starting point for future transactions and data management within the healthcare context.

5.3.1.3 The architecture behind the blockchain model

The development of the blockchain model was initiated using the Java programming language and the Spring Boot framework, providing a robust foundation for building a secure system for medical data storage.

Module 01 of the research lays the groundwork by introducing essential concepts and initial setups necessary for developing a secure data storage system. It emphasizes the architecture of the blockchain platform, detailing the design principles, data structures, and security mechanisms that facilitate the effective storage and retrieval of patient information.

The blockchain model is structured around the MVC (Model-View-Controller) architecture, which promotes separation of concerns and enhances maintainability. This approach involves developing various components, including models for representing data structures, controllers for managing application flow, and services for implementing business logic. Each of these components plays a crucial role in supporting the overall functionality of the blockchain platform, ensuring secure and efficient management of medical data.

By integrating these technologies and architectural principles, the blockchain model is designed to provide a reliable framework for handling sensitive patient information while leveraging the advantages of blockchain technology.

The methodology followed in the implementation process will be discussed in detail under Chapter 06 - Implementation.

5.3.1.4 Cross-Hash Validator Algorithm (A novel algorithm developed to securely store medical data)

5.3.1.5 Background & Motivation behind the development of the novel algorithm.

The Cross-Hash Validator is a fundamentally important algorithm embedded in the core of the blockchain model developed for securely storing medical data. This innovative algorithm addresses the limitations and vulnerabilities found in existing blockchain implementations, ensuring a more robust and reliable solution for managing sensitive healthcare information.
In our research in studying the existing implementation of the mechanisms storage of data in blockchain, I was able to identify the following flaws.

Data Duplication & Redundancy.

Blockchains are immutable, meaning that once data is recorded in a block, it cannot be altered or deleted. This characteristic ensures data integrity and security, as it prevents unauthorized modifications. However, to reflect any changes in the data, a new block must be added to the chain instead of modifying existing records.

While this immutability is beneficial for maintaining a reliable history of transactions, it can lead to significant redundancies and duplication of data within the blockchain. For instance, if a patient's information needs to be updated, such as a change in contact details a new block containing the updated information will be created. This results in multiple blocks containing similar or overlapping data, which can complicate data management and increase storage requirements within the blockchain system.

Lack of Query Optimization.

Another critical issue is the lack of query optimization when accessing data directly from the blockchain. The decentralized nature of blockchain technology often leads to time-consuming processes for querying data, as every transaction must be validated and traversed through multiple blocks. This inefficiency can hinder the responsiveness of applications relying on real-time access to patient data, making it challenging to meet the demands of healthcare providers and other stakeholders in the system.

To address the identified flaws in existing blockchain implementations, this innovative approach involves storing data and the blockchain separately.

In this model, rather than directly storing the entire data record on the blockchain, only a hash of the database record is stored in each block. This method significantly reduces redundancy and improves efficiency while maintaining the integrity of the data. To achieve this the afformention Cross- Hash validator algorithm plays a crucial role.

By storing data in a traditional database (such as MongoDB) and using the blockchain solely for securing the integrity of that data, we can mitigate issues associated with immutability and data duplication. Each time a record is created, updated, or deleted in the database, a unique hash is generated for that specific operation. This hash is then stored in the blockchain, representing the state of the data without duplicating the entire record.

For instance, consider a patient record that includes their name, age, and contact details. When a new patient is added to the database, a hash of the patient's information is generated and stored in the blockchain.

Hash for Patient Record:
data_hash: "a3b8b67e4d24f7cb3d48e0cd4c0344d6fa515f8f9e57ab2238c5a4d707ef3d1d
If the patient's contact details need to be updated later, a new hash is generated for the updated information and a new block is added to the blockchain, rather than modifying the existing block. This allows for a clear audit trail without creating redundancy

Hash for Updated Patient Record:
data_hash: "b4f56e7a74e1b34f2a75f03b1ef315cd71c4efb1b601ff25ed9bce012f3b1d9a"
This fundamental concept is explained thoroughly below and is achieved by the novel algorithm developed Cross-Hash Validator Algorithm.

5.3.1.6 The intuition behind CrossHashValidatorAlgorithm

The CrossHashValidatorAlgorithm is a critical component of the blockchain framework designed for securely managing medical data. The CrossHashValidatorAlgorithm stems from the CrossHashValidatorAlgorithm.java class and it consists of two main methods: hash and validator, each serving distinct but complementary purposes.

Hashing Method: The `hash` method is responsible for generating a SHA-256 hash from a given input string. This hashing process creates unique identifiers for the corresponding records that will be stored in the database. It also helps to generate unique identifiers for the blocks. The hashing method incorporates both vertical and horizontal hashing concepts, ensuring comprehensive validation of the data.

- **Vertical Hashing**: This involves generating hashes that link each block to its predecessor in the blockchain. It ensures the structural integrity of the blockchain by creating a chain of hashes where each block’s hash is based on its data and the hash of the previous block.
- **Horizontal Hashing**: This involves generating hashes for the individual records stored in the database. It ensures the integrity of the actual data by creating unique hashes for each database transaction (POST, PUT, DELETE), which are then stored in the blockchain.

2. Validator Method: The validator method performs comprehensive integrity checks on the blockchain. It implements two key validation mechanisms:

- **Vertical Validation**: This process verifies the integrity of the blockchain by ensuring that each block's previous hash matches the computed hash of its predecessor. This check maintains the structural integrity of the blockchain and prevents unauthorized modifications. The data that has been vertically hashed is validated through this process.
- **Horizontal Validation**: This method checks the consistency of the stored records by comparing the hashes of the current data against the hashes stored in the relevant blocks of the blockchain. It identifies the most recent transaction (using POST or PUT methods) for each document and ensures that the stored data matches the expected hash. The data that has been horizontally hashed is validated through this process.

By employing both vertical and horizontal validation, the CrossHashValidatorAlgorithm effectively safeguards the integrity and reliability of the medical data stored within the blockchain system. This dual-layer validation approach is essential for maintaining a secure and trustworthy healthcare data management environment.

A detailed explanation of the code and implementation of the algorithm will be discussed under Chapter - 6 Implementation of this paper.

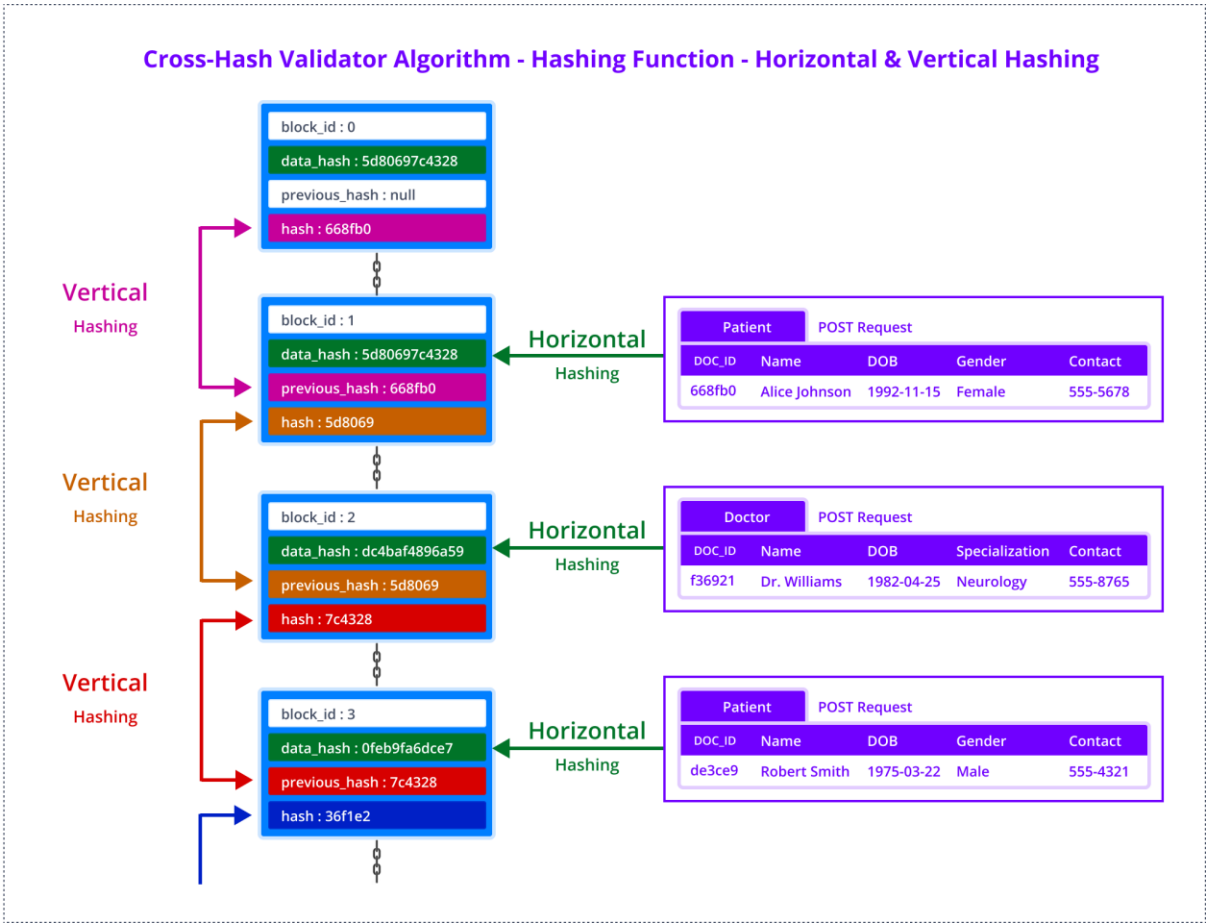


Figure 8: Cross-Hash Validator Algorithm - Vertical and Horizontal Hashing

5.3.1.7 The implementation of the CrossHashValidatorAlgorithm

The CrossHashAlgorithm was implemented under the utility classes of the project and the codebase for the implementation is given below.

```
package com.example.blockchainhealthcare.utils;

import com.example.blockchainhealthcare.model.BlockDTO;

import java.security.MessageDigest;
import java.util.*;

public class CrossHashValidatorAlgorithm {
    public static String hash(String input) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] hash = digest.digest(input.getBytes("UTF-8"));

            StringBuilder hexString = new StringBuilder();
            for (byte b : hash) {
                String hex = Integer.toHexString(0xff & b);
                if (hex.length() == 1) hexString.append('0');
                hexString.append(hex);
            }

            return hexString.toString();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public static String validator(ArrayList<BlockDTO> blockchain,
HashMap<String, String> records) {
        // Vertical Validation
        ArrayList<BlockDTO> blockchain_vertical_vld = new
ArrayList<>(blockchain);

        BlockDTO previous_block = blockchain_vertical_vld.get(0);
        int block_index = 1;

        while (block_index < blockchain_vertical_vld.size()) {
            BlockDTO block = blockchain_vertical_vld.get(block_index);
            String previous_block_hash_in_current_block =
block.getPreviousHash();
            String hash_of_the_previous_block =
CrossHashValidatorAlgorithm.hash(previous_block.toString());

            if
(!previous_block_hash_in_current_block.equals(hash_of_the_previous_block))
{
                return "Data integrity has been compromised in the
blockchain!";
            }

            previous_block = block;
            block_index++;
        }


        // Horizontal Validation
        ArrayList<BlockDTO> blockchain_horizontal_vld = new
ArrayList<>(blockchain);
```

```
        blockchain_horizontal_vld.remove(0);

        for (Map.Entry<String, String> record : records.entrySet()) {
            String record_document_id = record.getKey();
            String record_document_value = record.getValue();

            HashMap<String, BlockDTO> values = new HashMap<>();

            for (BlockDTO block : blockchain_horizontal_vld) {
                if (block.getDocument_id().equals(record_document_id)) {
                    values.put(block.getDocument_id(), block);
                }
            }

            Optional<Map.Entry<String, BlockDTO>> result =
values.entrySet().stream()
                    .filter(entry ->
entry.getValue().getHttp_method().equalsIgnoreCase("post") ||

entry.getValue().getHttp_method().equalsIgnoreCase("put"))
                    .max(Comparator.comparingInt(entry ->
entry.getValue().getBlock_id()));

            if (result.isPresent()) {
                if
(!result.get().getValue().getData_hash().equals(CrossHashValidatorAlgorithm
.hash(record_document_value))) {
                    return "Data integrity has been compromised! with
document_id: " + record_document_id;
                }

            }
        }
        return "Data is safe. Integrity not compromised!";
    }
}
```

5.3.1.8 The intuition behind the hash method in the CrossHashValidatorAlgorithm

Let's try to understand the operation of the operation behind the hash method in the Cross-Hash Validator Algorithm.

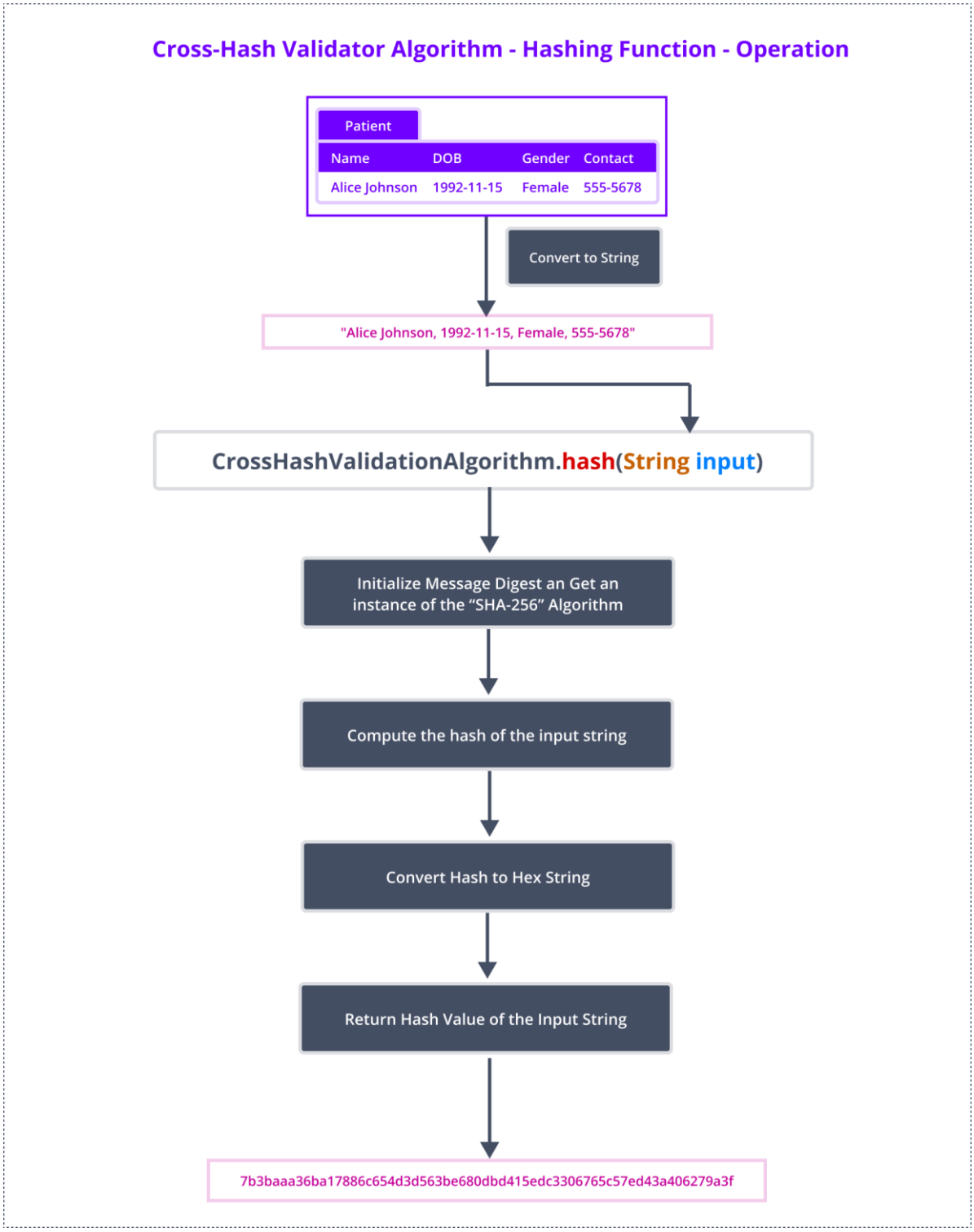


Figure 9: Cross-Hash Validator Algorithm - Hash Function Operation

Let's say you want to generate a hash value for a piece of information that you want to save in a database. In our approach prior to saving the data to the database, we need to save a digital footprint pertaining to that database transaction to a block in our blockchain under the field “data_hash.”

First, we take the data; according to the above example, the patient data pertaining to the patient named “Alice Johnson” is converted to a string format, which might look like this: "Patient Name: Alice Johnson, DOB: 1992-11-15, Gender: Female, Contact: 555-5678". We then feed this string as an input to the hash method of the CrossHashValidatorAlgorithm class.

As the output, we will obtain a stream of characters consisting of 64 hexadecimal digits, representing the unique hash for Alice Johnson's data. For example, the resulting hash might look like this:

```
7b3baaa36ba17886c654d3d563be680dbd415edc3306765c57ed43a406279a3f
```

This output will be stored in the “data_hash” field of the new block in the blockchain for the corresponding database operation made. By saving the hash in this field, we create a secure link between the on-chain data and the actual records stored in the database. This ensures that any changes in Alice’s data can be tracked through the blockchain while maintaining a clear and immutable history of transactions related to her medical records.

5.3.1.9 The intuition behind the hash method in the CrossHashValidatorAlgorithm – POST Request

In the previous example, we had a glimpse of an idea of how a record that we need to store in the database is converted to a data_hash using the hash method available in the CrossHashValidatorAlgorithm class.
Now let us delve deeper into a better illustrative example to understand this concept better.

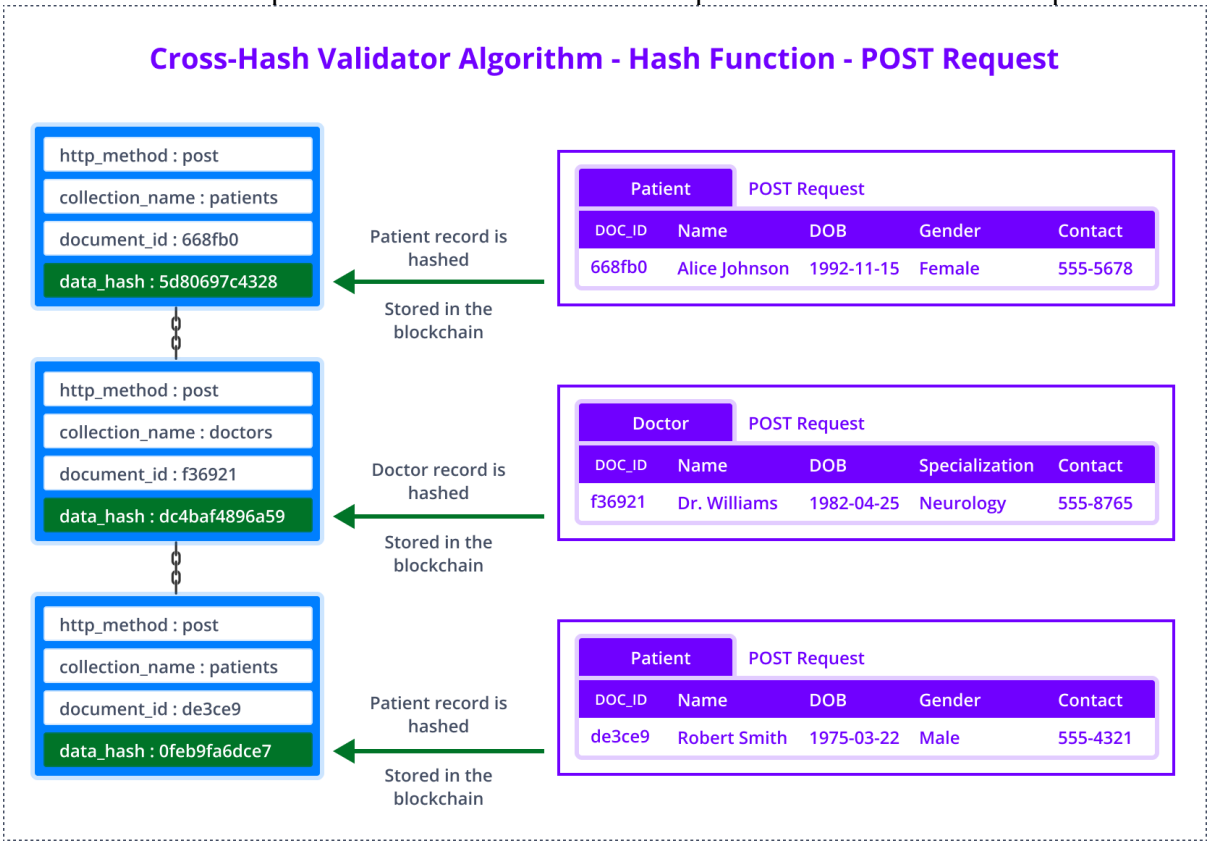


Figure 10: Cross-Hash Validator Algorithm - Hash Function Operation on a POST Request

As illustrated in the diagram, the patient named Alice is about to be stored in the database using a POST request. Before this data is saved, it is passed through the hash method in the CrossHashValidator class to generate a digital footprint. This hashed value is then stored in a block of the blockchain under the data_hash field.

Subsequently, two additional POST requests were made for Dr. Williams and patient Robert. The corresponding data_hash values for these records have been appropriately generated and recorded in their respective blocks within the blockchain. This process ensures that each piece of data has a unique and secure identifier, maintaining the integrity and security of the medical information stored.

5.3.1.10 The intuition behind the hash method in the CrossHashValidatorAlgorithm – PUT Request

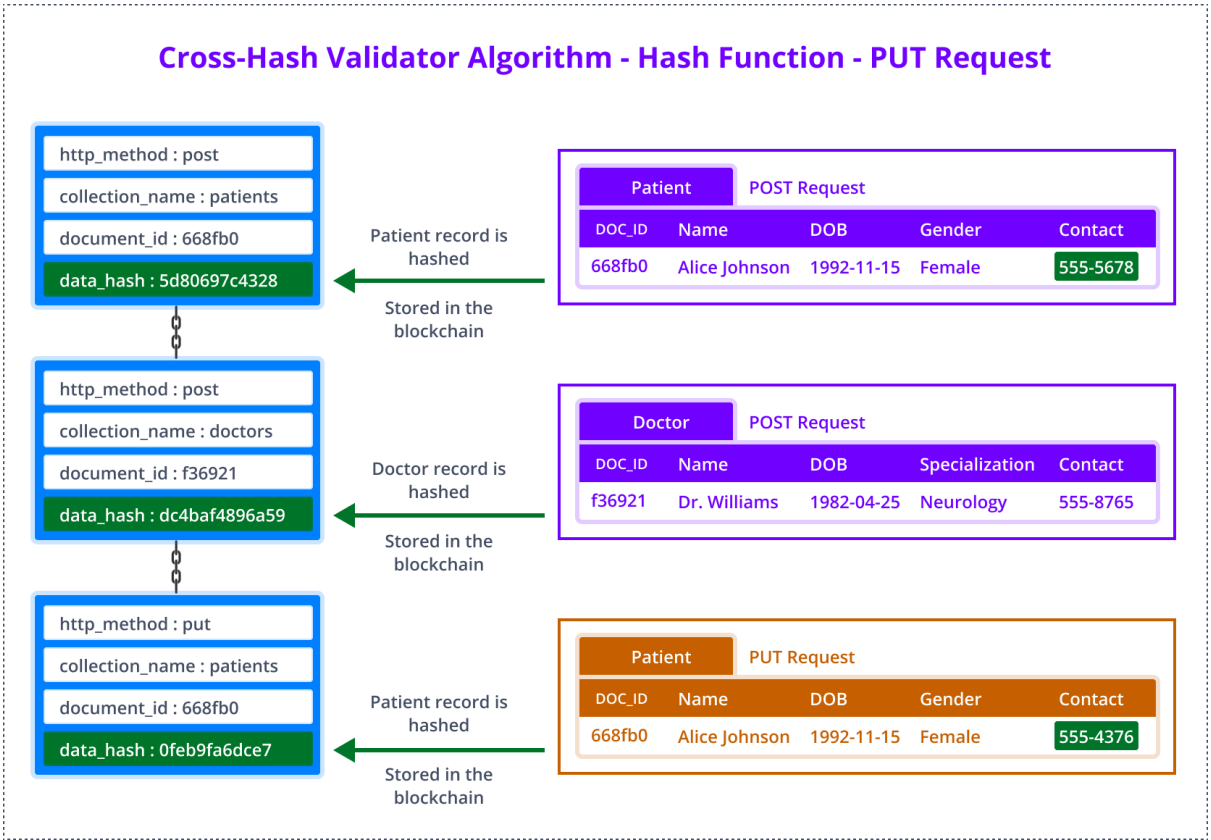


Figure 11: Cross-Hash Validator Algorithm - Hash Function Operation on a PUT Request

As illustrated in the diagram, the patient named Alice is about to be stored in the database using a POST request. Before this data is saved, it is passed through the hash method in the CrossHashValidator class to generate a digital footprint. This hashed value is then stored in a block of the blockchain under the data_hash field.

Following this, the details pertaining to Dr. Williams are also hashed and stored in the corresponding block of the blockchain during the subsequent POST API call.

Now, consider a scenario where patient Alice has acquired a new SIM card and changed her contact number. To handle this update, the database must reflect the new contact information. When such an update occurs, the revised record for Alice is processed through the hash method again and stored in a new block within the blockchain. As illustrated in the diagram, the collection_name and document_id remain the same, while the http_method is updated to PUT, and the data_hash is recalculated to accurately reflect the changes. This ensures that every update is securely documented, maintaining the integrity and traceability of the medical data.

5.3.1.11 The intuition behind the hash method in the CrossHashValidatorAlgorithm – DELETE Request

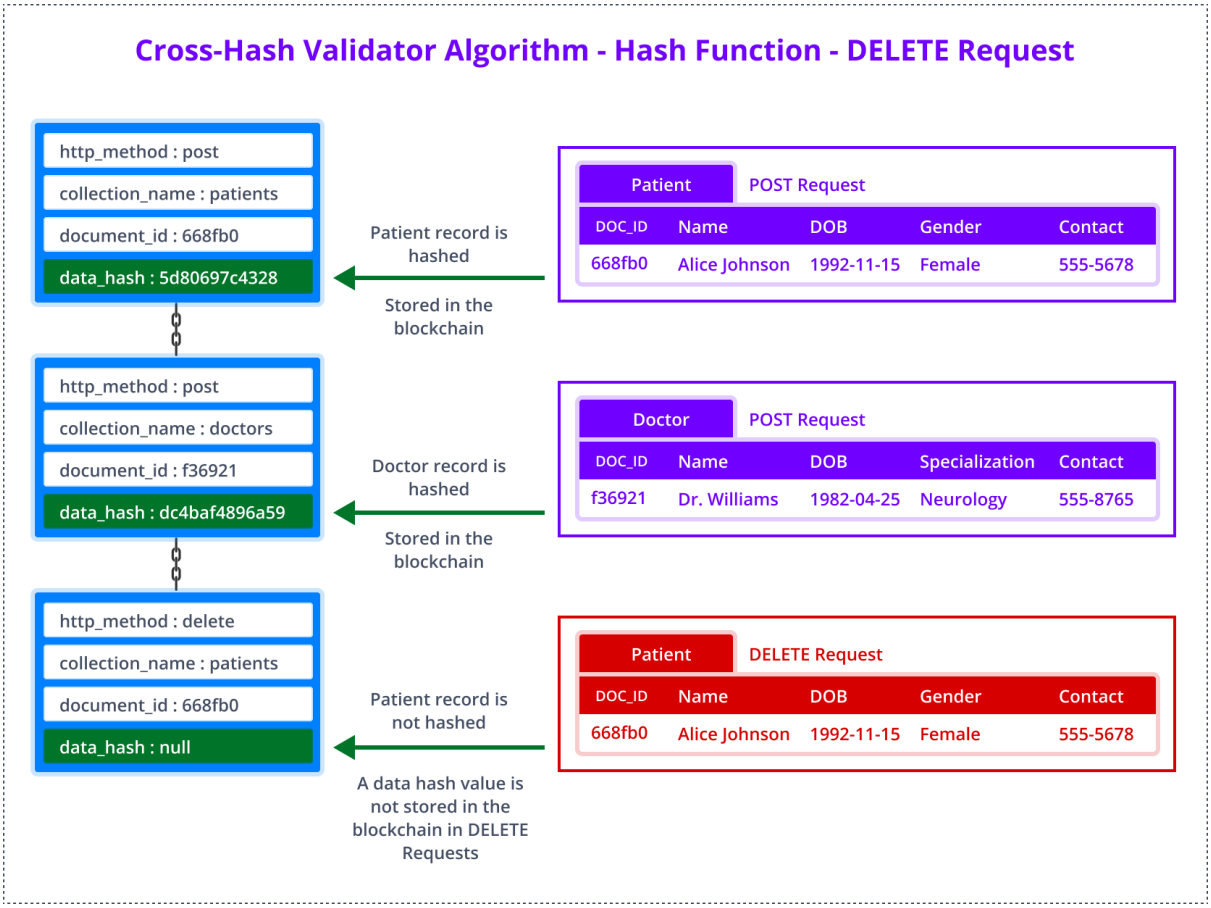


Figure 12: Cross-Hash Validator Algorithm - Hash Function Operation on a DELETE Request

As illustrated in the diagram, the patient named Alice is about to be stored in the database using a POST request. Before this data is saved, it is passed through the hash method in the CrossHashValidator class to generate a digital footprint. This hashed value is then stored in a block of the blockchain under the data_hash field.

Following this, the details pertaining to Dr. Williams are also hashed and stored in the corresponding block of the blockchain during the subsequent POST API call.

Assume that the details pertaining to Alice are no longer needed in our database. To remove her record, a DELETE API request must be made. However, it is essential for the blockchain to also reflect this deletion. To capture this change, a new block will be added to the blockchain with the http_method set to DELETE for the specific document_id and collection_name. Since we are deleting data, no data hash will be generated, and the data_hash field in this block will be set to null. This ensures the blockchain accurately records the deletion of the record while maintaining its integrity and traceability.

5.3.1.12 The intuition behind the hash method in the CrossHashValidatorAlgorithm – GET Request

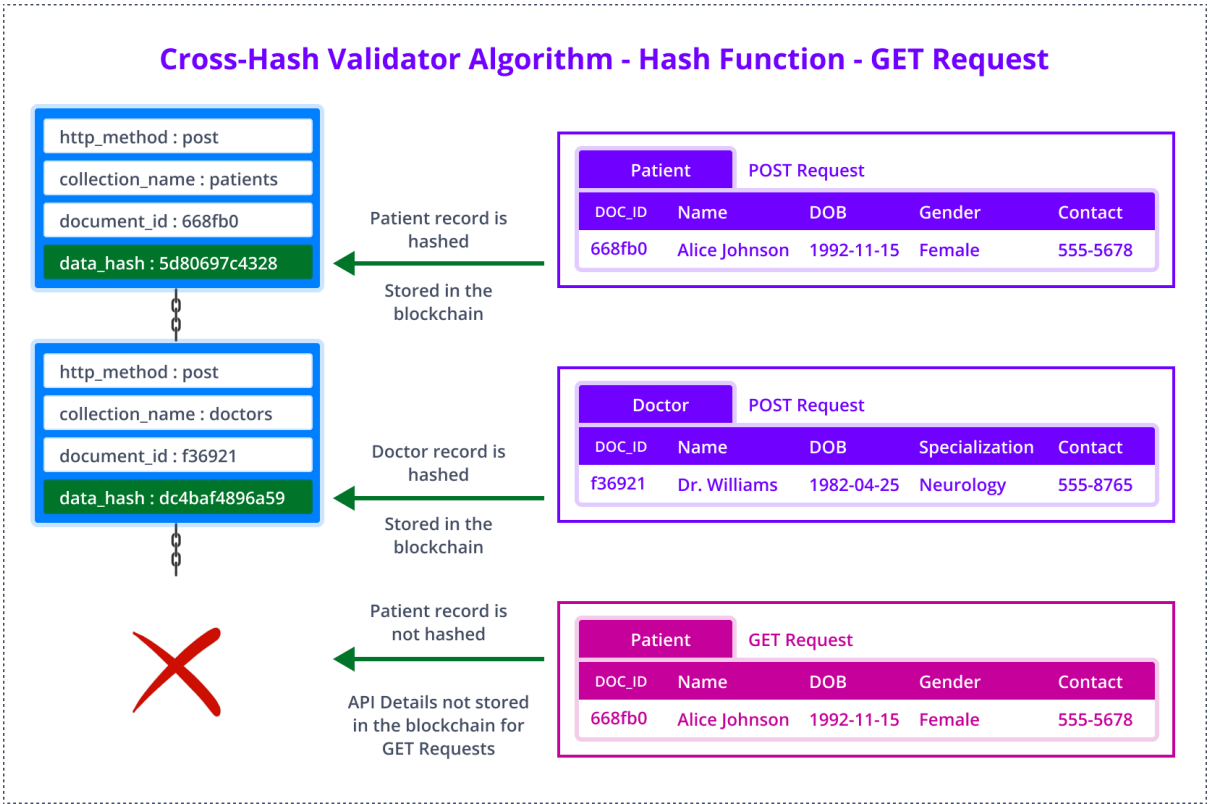


Figure 13: Cross-Hash Validator Algorithm - Hash Function Operation on a GET Request

As illustrated in the diagram, the patient named Alice is about to be stored in the database using a POST request. Before this data is saved, it is passed through the hash method in the CrossHashValidator class to generate a digital footprint. This hashed value is then stored in a block of the blockchain under the data_hash field.

Following this, the details pertaining to Dr. Williams are also hashed and stored in the corresponding block of the blockchain during the subsequent POST API call.

Now, let's assume a user needs to access the data pertaining to Alice Johnson. For this, the user would make a GET API call. Unlike POST, PUT, and DELETE requests, the details of GET requests are not stored in the blockchain. This is because the data recorded from POST, PUT, and DELETE calls is sufficient to determine the integrity and validity of the data. Additionally, storing every GET request would unnecessarily lengthen the blockchain, making it inefficient and time-consuming to traverse through the blocks. By only recording data-altering operations, we maintain a streamlined and efficient blockchain while ensuring data integrity.

5.3.1.13 The intuition behind the Horizontal hashing and Vertical hashing terminology

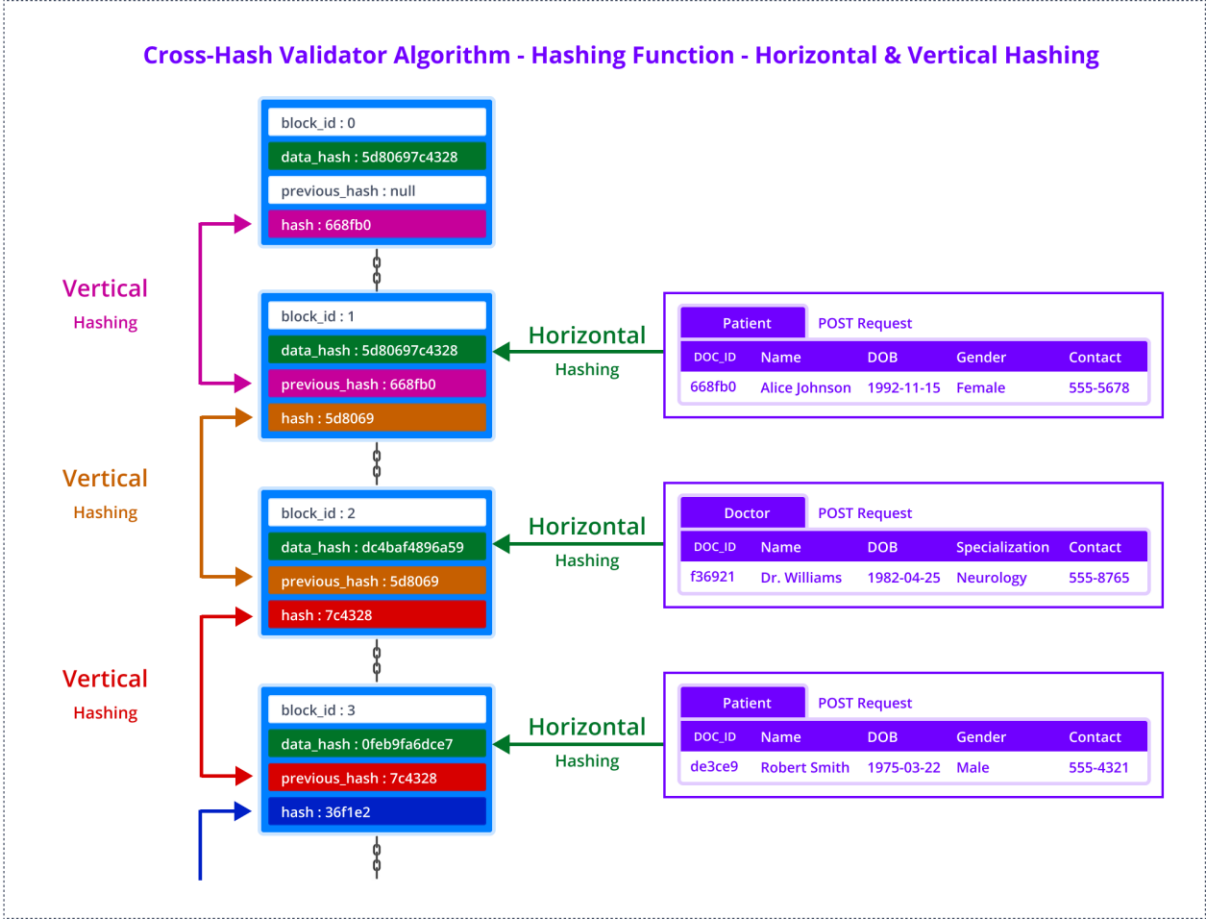


Figure 14: Cross-Hash Validator Algorithm - Vertical and Horizontal Hashing

You may have come across the diagram earlier in this report, but here it will be explained in greater detail.

The concept of storing a data_hash for each record in the database resulting from POST, PUT, and DELETE API calls in a block of the blockchain is known as **"Horizontal Hashing."** This terminology is used because, for every record manipulated in the database, a corresponding block is added to the blockchain. This operation is analogous to a horizontal action, hence the name.

Every block is connected to its previous block using the previous hash, establishing the integrity of the chain through this connection. As seen in the diagram, we can observe operations occurring along the chain in the vertical direction, which is referred to as **"Vertical Hashing."**

The terms "Vertical Hashing" and "Horizontal Hashing" were intentionally chosen to avoid confusion between these two operations and to easily identify each operation separately.

5.3.1.14 The intuition behind the validator method in the CrossHashValidatorAlgorithm

In the previous section, we conducted a comprehensive analysis of the hash method within the CrossHashValidator Algorithm. In this section, we will delve into the details of the validator method of the same class.

The hash and validator methods function together as essential components of the algorithm, supporting each other's core objectives. While the hash method is responsible for generating hashed values, the validator method ensures data integrity by utilizing these hashed values to perform thorough checks on the blockchain.

The validator method performs comprehensive integrity checks on the blockchain. It implements two key validation mechanisms:

- **Vertical Validation**:
    - This process verifies the integrity of the blockchain by ensuring that each block's previous hash matches the computed hash of its predecessor. This check maintains the structural integrity of the blockchain and prevents unauthorized modifications.
    - The data that has been vertically hashed is validated through this process.
- **Horizontal Validation**:
    - This method checks the consistency of the stored records by comparing the hashes of the current data against the data_hashes stored in the relevant blocks of the blockchain. It identifies the most recent transaction (using POST or PUT methods) for each document and ensures that the stored data matches the expected hash.
    - The data that has been horizontally hashed is validated through this process.

Refer to the following diagram for more information.

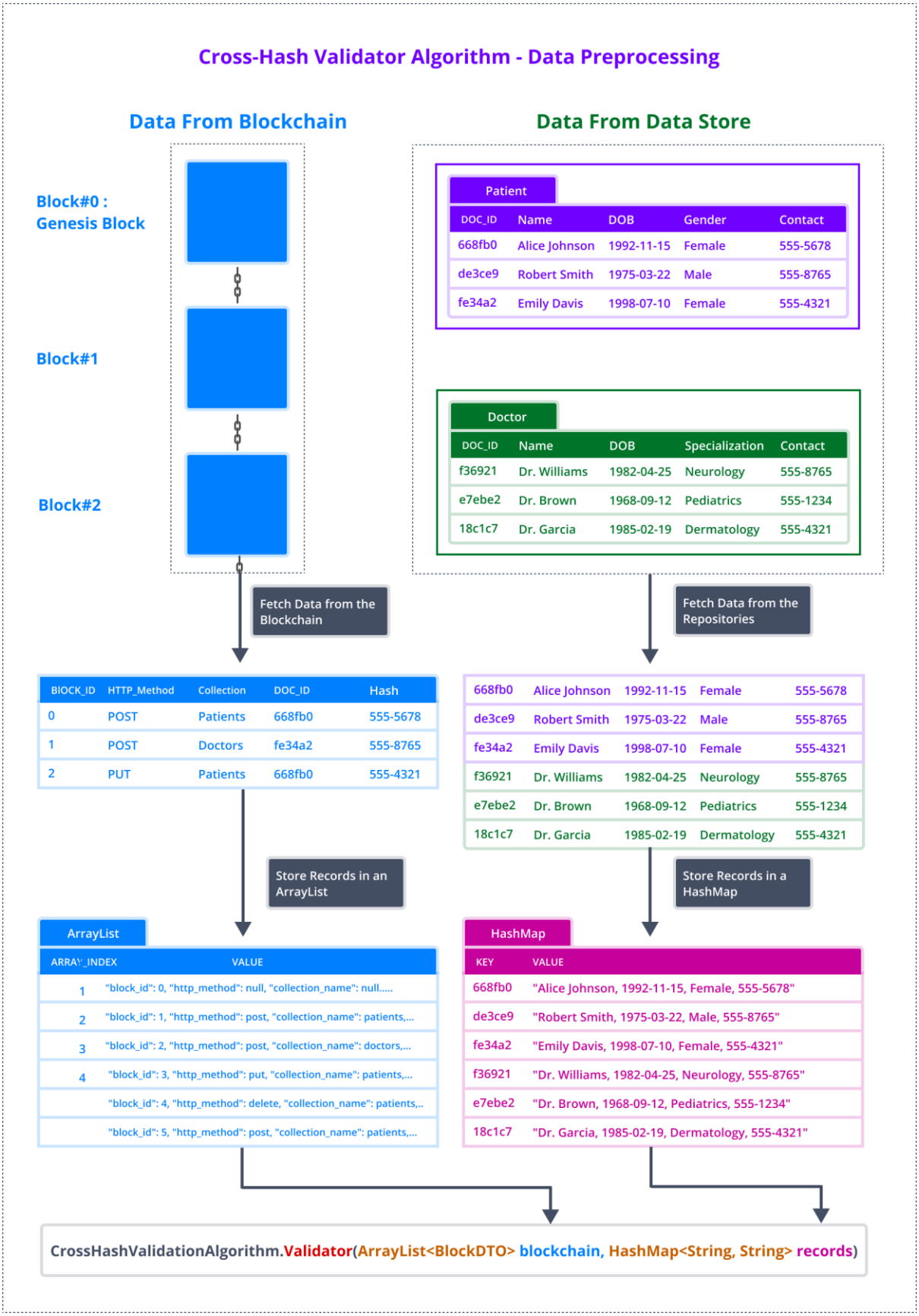


Figure 15: Cross-Hash Validator Algorithm - Validator Method - Data Preprocessing

Stage 01: Cross-Hash Validator Algorithm - Validator Method - Data Preprocessing

Before performing validation to assess data integrity, the data must be preprocessed into a suitable format for the algorithm. We will be receiving data from two sources: the blockchain, which includes the blockchain data and its corresponding blocks, and the database that holds medical information, such as records for patients, doctors, insurance agencies, and lab reports. For the following example, we will focus on a database that stores patient and doctor data for simplicity.

The validator method in the CrossHashValidator class accepts two arguments: the blockchain and the medical records. The blockchain should be provided as an ArrayList, while the medical records must be supplied in the format of a HashMap.
The medical records retrieved from the data store are stored in a HashMap before being passed as an argument to the validator method. This HashMap consists of key-value pairs, where the key is the corresponding document_id of the medical record, and the value is a string representing the medical record, as illustrated in the diagram above.
The array list of blockchain and the HashMap of the medical records thus fed into the validator method

Stage 02: Cross-Hash Validator Algorithm - Validator Method - Vertical Validation
After the data has been pre-processed properly into a suitable format, the data will be the vertical hashing process and the horizontal hashing process will be validated via vertical validation and horizontal validation process respectively. Out of the two validation processes, vertical validation will take precedence and will be the next step followed by the data pre-processing stage. The reason for its precedence is that we want to assess the viability and the integrity of the data assessed in the blockchain prior to using the data_hash of blocks in the blockchain to assess the viability and integrity of the medical records stored in the database.
Vertical validation is done to determine the integrity of the blockchain by ensuring that each block's previous hash matches the computed hash of its predecessor. This check maintains the structural integrity of the blockchain and prevents unauthorized modifications.
The following indicates the key steps in the vertical validation process.

1. Initialize Variables for Vertical Validation

The validation begins by initializing two key variables:

- **previous_block**: This variable holds the first block of the blockchain, serving as the starting point for comparison.
- **block_index**: This integer variable is initialized to 1, indicating the current position in the blockchain as we iterate through the blocks.

2. Iterate Over Blockchain for Vertical Validation

A loop is established to traverse through the blockchain starting from the second block (index 1) up to the last block. This allows for a systematic check of each block against its predecessor.

3. Extract Previous Hash from the Current Block and Compute Hash of Previous Block

For each block in the iteration, the following actions are performed:

- The **previous hash** is extracted from the current block using block.getPreviousHash().
- The **hash of the previous block** is computed by calling the hash method on the previous_block object, which generates a hash string representation of the previous block's data.

4. Compare Hashes for Vertical Validation

The computed hash of the previous block is then compared with the extracted previous hash from the current block. This comparison is essential for validating the integrity of the blockchain.

5. Check for Hash Mismatch

If the hashes do not match, it indicates a potential integrity issue within the blockchain. In this case, the method returns the message: **"Data integrity has been compromised in the blockchain!"** This serves as a warning that the chain has been altered or corrupted in some manner.

If the hashes match, it confirms that the connection between the blocks is valid, and the process proceeds to the next phase, which is horizontal validation.

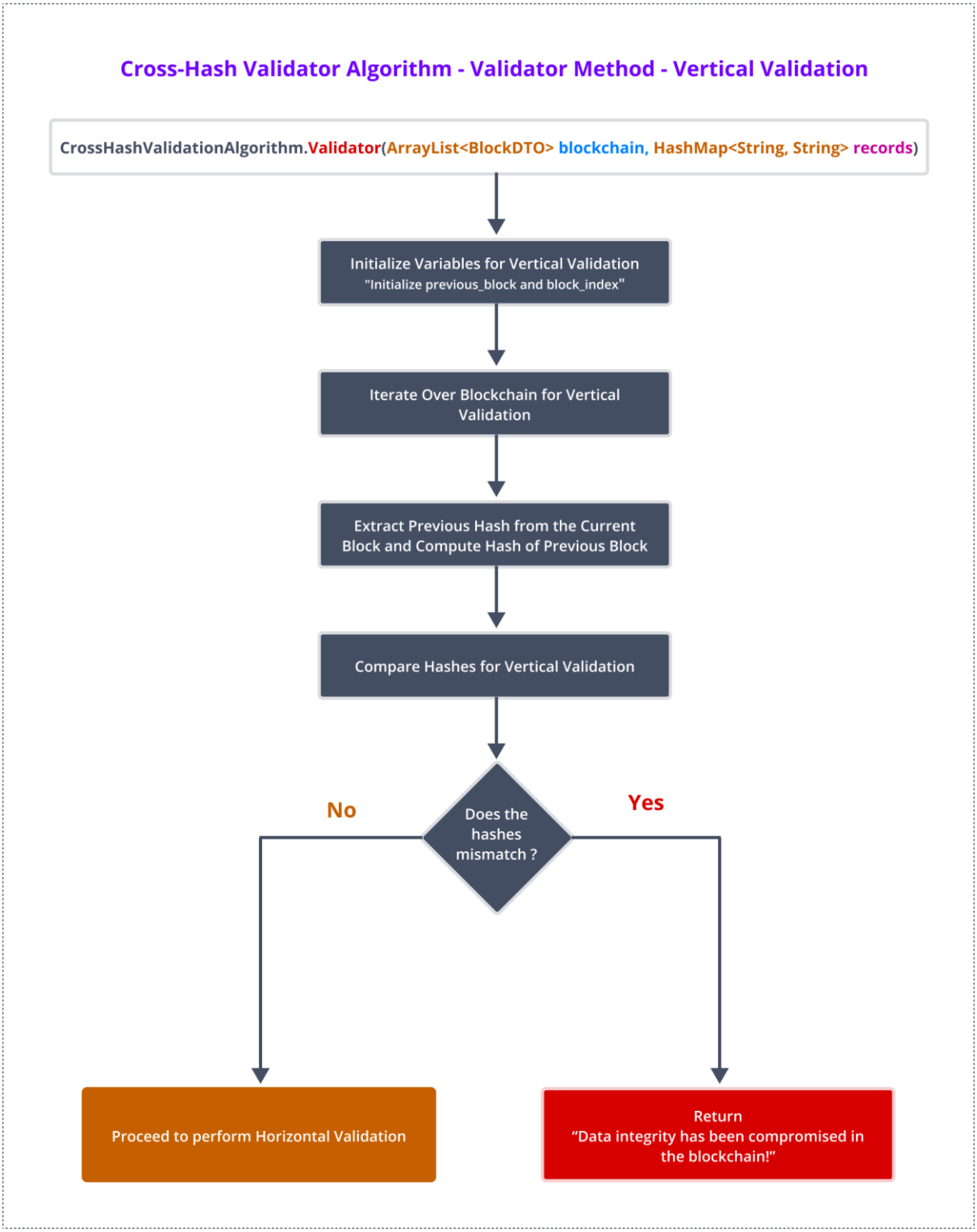


Figure 16: Cross-Hash Validator Algorithm - Validator Method - Vertical Validation

Stage 03: Cross-Hash Validator Algorithm - Validator Method - Horizontal Validation

If the algorithm does not encounter any hash mismatches pertaining to previous_hash stored in a block and the hash of the previous block, it ensures the integrity of the data stored in the blockchain. Thus, can be used to assess the integrity of the medical records stored in the database.

The horizontal validation is performed to check the consistency of the stored records by comparing the hashes of the current data against the data_hashes stored in the relevant blocks of the blockchain. It identifies the most recent transaction (using POST or PUT methods) for each document and ensures that the stored data matches the expected hash. The data that has been horizontally hashed is validated through this process.

The following indicates the key steps in the vertical validation process.
1. Initialize Variables for Horizontal Validation

The horizontal validation process begins by initializing necessary variables:

- **blockchain_horizontal_vld**: This list contains copies of blocks from the blockchain, excluding the genesis block (first block), as it's not involved in data storage.
- **records**: This hashmap contains medical records fetched from the database, where each record is represented by a document ID and its corresponding value.

2. Iterate Over Records

A loop iterates through each entry (document ID and value) in the records hashmap retrieved from the database. This allows for a comparison between the stored blockchain data and the current database records.

3. Find Matching Blocks

For each record in the iteration, the validator identifies blocks in blockchain_horizontal_vld that match the document ID of the current record. These matching blocks are stored in a hashmap named values.

4. Create a Map of Blocks that Match the Document ID

Within the blockchain_horizontal_vld, blocks associated with the same document ID as the current record are stored in the values hashmap. This step prepares for the subsequent validation of the most recent transaction related to that document ID.

5. Find Latest Valid Block

The validator proceeds by finding the block within values that represents the latest valid transaction (either a 'post' or 'put' HTTP method). This is achieved using Java Stream operations to filter and identify the block with the highest block_id among eligible transactions.

6. Validate the Latest Valid Block

Once the most recent valid block is identified:

- The data_hash of this block is compared with the hash of the current record's value using the hash method from the CrossHashValidatorAlgorithm.
- If the hashes do not match, indicating a discrepancy or potential tampering with the data, the method returns: **"Data integrity has been compromised! with document_id: xxxxxx"**

7. Data Integrity Check Result

If all records pass the validation without any mismatches, the method concludes with the message: **"Data is safe. Integrity not compromised!"**

5.3.1.15 Benefits and Drawbacks of the CrossHashValidator Algorithm

The CrossHashValidator Algorithm offers a novel approach to enhancing data integrity and security within blockchain implementations. By focusing on minimizing data redundancy and streamlining querying processes, this algorithm addresses some of the critical challenges faced by traditional blockchain systems.
However, like any innovative solution, it comes with its own set of benefits and drawbacks. This section will explore the advantages and limitations of the CrossHashValidator Algorithm, providing a comprehensive evaluation of its performance compared to typical blockchain algorithms.

Benefits

1. **Enhanced Data Integrity**
    - **Description**: The CrossHashValidator algorithm incorporates both vertical and horizontal validation, ensuring robust integrity checks that effectively detect any unauthorized changes to the blockchain.
    - **Comparison**: Unlike traditional algorithms that may only perform basic checks, this dual-layer validation significantly improves data security. This is achieved by hashing operation that is performed in both vertical and horizontal directions.
2. **Separation of Data and Hashes**
    - **Description**: By storing only hashes in the blockchain and keeping actual data in a separate database, the algorithm reduces redundancy and potential storage issues within the blockchain.
    - **Comparison**: Traditional approaches often lead to data duplication and larger blockchain sizes, making them less efficient.
3. **Efficient Hashing Process**
    - **Description**: Utilizing SHA-256 for hashing ensures a high level of security and uniqueness for each data entry.
    - **Comparison**: Many blockchain algorithms use similar hashing methods, but the implementation of a dedicated hashing function for specific use cases enhances reliability.
4. **Simplified Querying**

- Users can directly query the database to obtain data, rather than querying the blockchain. This results in faster and more efficient data retrieval.

Drawbacks

1. **Increased Complexity**
   - The algorithm's dual validation process introduces additional complexity, which may require more resources and expertise to implement and maintain. Simpler blockchain algorithms might be easier to implement and manage, especially for small-scale applications.
2. **Performance Overhead**
   - The thorough validation process may lead to increased computational overhead, potentially slowing down performance with large datasets.
   - Some traditional algorithms may sacrifice validation rigor for speed, making them more suitable for high-throughput environments.
3. **Scalability Challenges**
   - As the size of the blockchain grows, the complexity of validations can lead to scalability issues, especially if not optimally designed.Some blockchain solutions are designed specifically for scalability, handling larger datasets with ease.
4. **Limited Adoption**
   - **Description**: Being a novel approach, the CrossHashValidator may face challenges in adoption within existing systems that rely on more conventional blockchain algorithms.Established algorithms may have broader community support and documentation, making them easier to adopt.

5.3.1.16 UI Design and Frontend Development

The idea is to develop a web application that helps users to seamlessly interact with the blockchain platform that we designed to securely store medical data. The frontend of the web application was designed using ReactJS technology. The development will be discussed further under Chapter 06: implementation.

5.3.1.17 Summary

Module 01 establishes the foundation for securely storing medical data using blockchain technology, leveraging Java and Spring Boot. This module focuses on designing a robust architecture for patient data management, detailing the design principles, data structures, and security mechanisms.

Blockchain Architecture

- Utilizes MVC architecture, incorporating models, controllers, services, and utility classes.
- Detailed exploration of the blockchain platform's architecture for efficient and secure data storage and retrieval.

Cross-Hash Validator Algorithm

- **Hash Function:** Implements the SHA-256 hashing algorithm for unique and secure block data representations, incorporating both vertical and horizontal hashing for comprehensive validation.
    - **Vertical Hashing:** Links each block to its predecessor, ensuring structural integrity.
    - **Horizontal Hashing:** Generates unique hashes for each database transaction, stored in the blockchain.
- **Validation Function:** Ensures data integrity through vertical and horizontal validations.
    - **Vertical Validation:** Sequentially checks the integrity of each block, verifying that each block's previous hash matches the actual hash of the preceding block.
    - **Horizontal Validation:** Cross-references blockchain entries with current records to prevent tampering.

Technical Aspects and Evaluation

- Detailed discussion of the algorithm's design, implementation, and technical methodologies.
- Evaluation of the benefits and drawbacks of the algorithm.

UI Design & Development

- Focuses on creating a user-friendly and intuitive interface for the blockchain platform.

Highlights

- Establishing the blockchain architecture with Java and Spring Boot.
- Comprehensive overview of MVC-based models, controllers, services, and utility classes.
- Introduction and detailed discussion of the Cross-Hash Validator Algorithm to enhance data security and integrity.
- Technical evaluation of the algorithm's implementation.
- User-friendly UI design for the blockchain platform.

5.3.2 - Module 02: Enhancing Security, Scalability and Reliability of the Blockchain Model by Creating a Custom Blockchain Network and Consensus Algorithm

5.3.2.1 Introduction

Module 2 focuses on enhancing the security, scalability, and reliability of the blockchain model that we proposed for the healthcare sector. As highlighted in existing literature, traditional healthcare IT infrastructure faces significant challenges, such as vulnerability to cyberattacks, data breaches, and scalability issues. The need for robust security measures, scalability enhancements, and improved reliability has driven the exploration of blockchain technology in the healthcare sector.

This research module focuses on advancing the architecture of blockchain models to address the existing security and privacy concerns while overcoming scalability limitations and deactivate nodes issues in current existing blockchain networks. Conventional healthcare systems often rely on centralized databases, making them susceptible to cyber threats, ransomware attacks, and unauthorized access. By leveraging the unique features of blockchain, including encryption, immutability, and decentralized consensus mechanisms, we aim to fortify the security of healthcare data, ensuring that patient information remains confidential and tamper-proof.

5.3.2.2 Consensus Algorithm

Blockchain is a decentralize system, so in this decentralize system there is no centralize server to monitor all transactions and what happen in the system. Therefore, we need a algorithm to do valid transactions and record valid transactions only to the blockchain.[36] For that we need a consensus. That is a huge challenge when it comes to the blockchain.[36]

Consensus Algorithm considered as the core of each blockchain.[36] In decentralized systems consensus is a problem that who can add the next block to the blockchain or who are allowed to add blocks to the blockchain. So, in this healthcare context usually there are limited number of nodes available when it comes to a Hospital or a medical center. And when it comes to healthcare sector, data privacy, data accuracy is very important and directly affect to the patients.

Since the proposed blockchain is a permissioned private blockchain that designed for hospitals or medical centers, I choose the most efficient, scalable, and secure existing consensus algorithm and I improved it to match for the healthcare sector. Chosen consensus algorithm is Proof of Authority (PoA) and it has several drawbacks as well. There are various many existing consensus algorithms that are discussed in literature review of module 02.

The consensus mechanism plays a crucial role in determining how transactions are validated and added to the blockchain. The choice of consensus mechanism can have a direct impact on the security and the scalability of a blockchain network. So, in this module one of the main tasks is to research and suggest a best custom consensus algorithm for the blockchain model which are proposed for the healthcare sector.

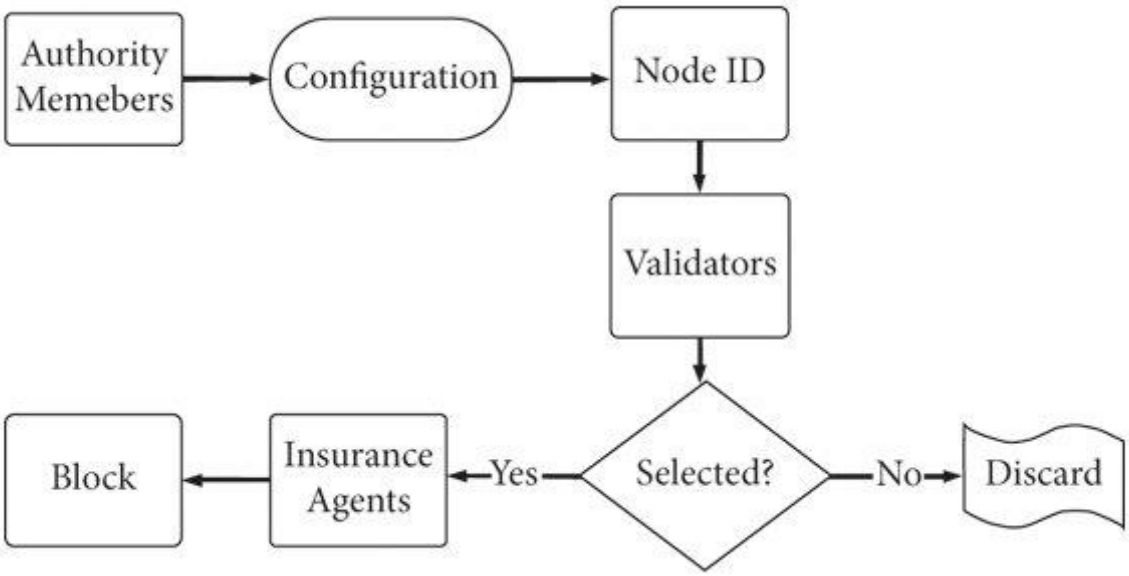


Figure 17: Consensus Algorithm (PoA – Proof of Authority) [63]

The Figure 18 shows how mostly use PoA (Proof of Authority) consensus algorithm works in graphically.
Considering the healthcare sector, it's better to have system to validate which parties can mine block for the blockchain, otherwise blockchain can be filled with a lot of fake health data which is not good for the quality of this blockchain system. Therefore, Proof of Authority is best fit for that scenario.
Because as you can see in figure 18, In Proof of Authority algorithm there are pre-defined validators who can add blocks to the network. Which means there are nodes who can add blocks to the blockchain and there are nodes who cannot add blocks to the blockchain. As you can see in the diagram there are authority members which are hospital authorities in this context. So, they can choose which nodes are the validators and those nodes can add blocks to the blockchain. Above diagram shows Authority Members decides whether that Node is a validator or not, If not then that node cannot add block and dicarded. If yes then that node can add a block.

This is how Proof of Authority (PoA) works and let's discuss pros and cons of this algorithm and why we need to improved this algorithm when it comes to the blockchain concept in healthcare sector.

Pros	Cons
Faster transaction processing times compared to other consensus algorithms	More centralized compared to other consensus algorithms
More secure and lower 51% attack risk, since authority nodes are trusted	Network security relies on authority nodes
Less energy consumption	Authority nodes can be changes with the time.
Easy decision making and governance the network	

So, you can see the proof of authority algorithm is more perfect consensus algorithm when it comes to the security, reliability and scalability. But it has several cons like more centralized compared to the other consensus algorithms.

That's because authority nodes decided who will be the validators and who are not also authority nodes can remove validators and add validators anytime. Because of that Network security and blockchain security also relies on authority nodes.

Novelty and Innovation

To address those issues, we introduce delegated voting system to the PoA algorithm. Which is a periodic voting system to clear potential malicious validators and secure the blockchain network.

Then the algorithm and network will be more decentralized and Network security no more relies only on authority nodes.

There are so many consensus algorithms which have combined with delegated voting systems, but still not exist for the Proof of Authority (PoA) algorithm.
We can call it **"Proof of Accountability Voting (PoAV)"**

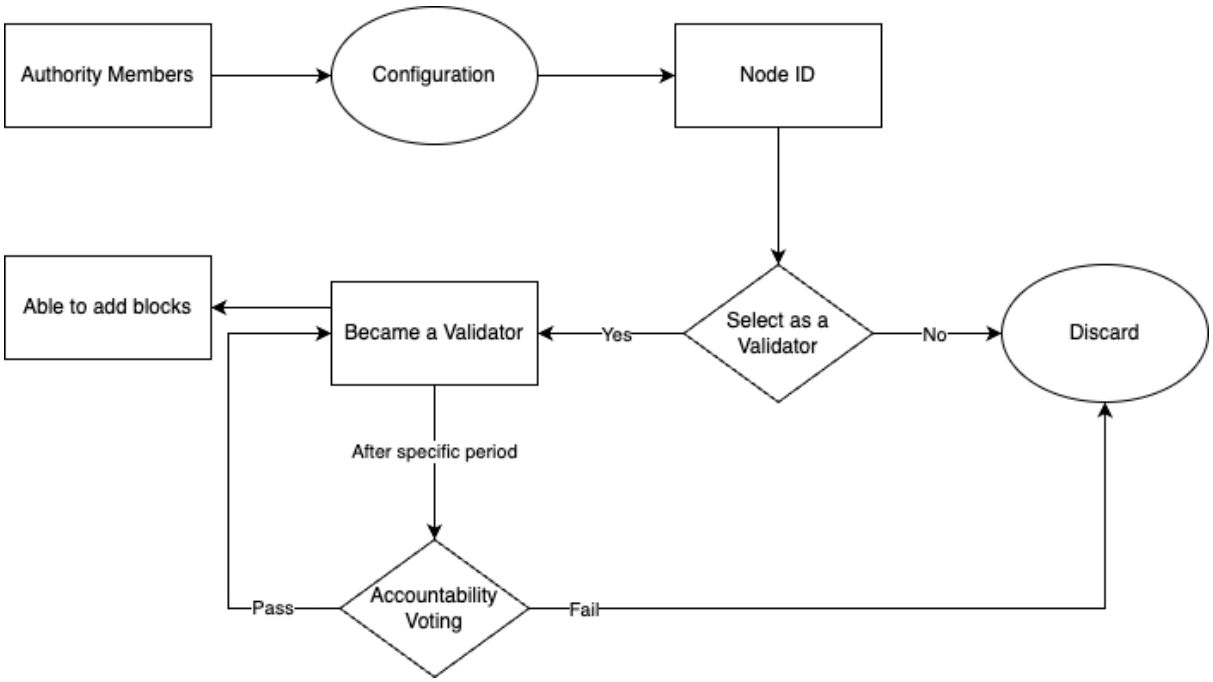


Figure 18: Custom Consensus Algorithm (PoAV – Proof of Accountability Voting)

Here as you can see, first steps same as the Proof of Authority (PoA) Consensus algorithm and then after specific period, there will be a voting assessment to check and clear the malicious validators from the network.
In this voting, every node can participate, in healthcare sector

Let's think a doctor as a validator and he or she not a good doctor since past month, then patients can vote and remove from the system as a validator, ***in this improved algorithm we addressed to the major issues that have in Proof of Authority (PoA) algorithm***, which are.

1. Centralization and validators and network control by authority nodes
2. Network security and governance relies on authority nodes.

5.3.2.3 Distributed Network with enhancing Scalability

Novelty and Innovation

This proposed customised blockchain network uses a gossip protocol-based consensus mechanism, providing various advances to address scalability and reliability challenges common in existing blockchain implementations.

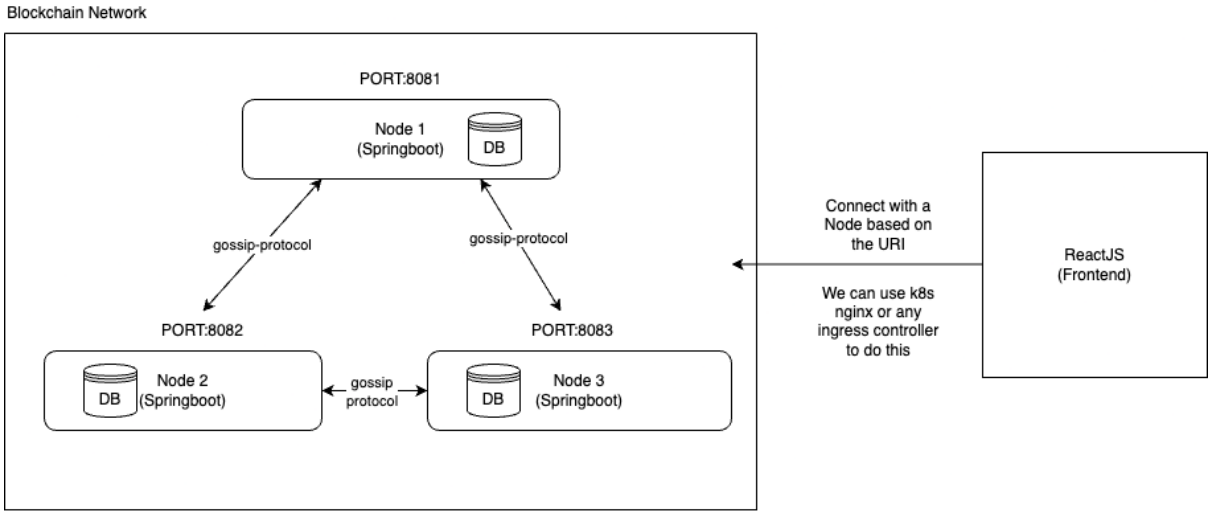


Figure 19: Blockchain Network

This is a simple network which has only three nodes, so here every node monitoring and each other nodes in the network. We can implement this network in the healthcare context because when it comes to a healthcare sector hospital or medical center, they don't need large number of nodes. That advantage we can take to the blockchain network to develop a robust network.

Addressing Scalability and Reliability

5.3.2.3.1.1 Scalability

1. Throughput Improvement:

- Our network design maximises throughput by allowing transactions to be processed in parallel. The gossip protocol shortens transaction propagation times, allowing for faster consensus.

2. Latency Reduction:

- The gossip protocol enables the quick transmission of transactions and blocks, drastically lowering the time required for a transaction to be confirmed.
- The adaptive synchronisation technique significantly minimises latency by rapidly reintegrating nodes into the network following failures.

3.Handling Large Network Size:

- The protocol is intended to scale effectively with the number of nodes. It accomplishes this by utilising decentralised communication patterns that evenly spread the workload across the network.
- Hierarchical topologies keep the network efficient even as the number of nodes increases.

5.3.2.3.1.2 Reliability

1.Robust Node Monitoring:

- Continuous monitoring of node status guarantees that any node failure is noticed quickly and appropriate measures are implemented to maintain network integrity.
- The adaptive synchronisation approach ensures that nodes can quickly catch up with the network state during recovery, avoiding data inconsistencies.

2.Consistency and Fault Tolerance:

- The network design assures that all nodes, even those that fail, have a consistent view of the blockchain data.
- The gossip protocol improves fault tolerance by guaranteeing that changes are efficiently transmitted, hence preventing a single point of failure.

5.3.2.4 Summary

Module 02 focuses on improving the security, scalability, and reliability of a blockchain exclusively for the healthcare industry. Traditional healthcare IT systems suffer threats such as cyberattacks, data breaches, and scalability concerns. This study seeks to address these issues by developing blockchain architecture and utilising capabilities such as encryption, immutability, and decentralised consensus methods to protect patient data.

The consensus algorithm is a the core component of this module, as it is required for validating transactions in a decentralised system, scalability and security of the blockchain system. The chosen Proof of Authority (PoA) algorithm, designed specifically for permissioned private blockchains in hospitals, provides data confidentiality and correctness. The study looks into how to improve PoA to better meet healthcare demands, address its disadvantages, and improve security and scalability and developed a new improved consensus algorithm called “Proof of Accountability Voting (PoAV)”.

Additionally address the issue of existing blockchain networks which is the inability of nodes to monitor their neighbors continuously, especially after a node failure and recovery. By using the advantage of limit number of nodes needed in healthcare sector, address that issue by developing custom blockchain network using gossip protocol.

5.3.3 - Module 03 - Streamlining Patient Data Consent Management Processes

5.3.3.1 Introduction

The requirement for effective and secure patient data consent management has grown in importance due to the constantly changing nature of the healthcare industry. Because patient information is sensitive, it is essential to healthcare ethics to guarantee that patients have a choice over how their information is used. Traditional methods of obtaining and managing patient consent heavily rely on the knowledge and efficiency of on-site staff, their access to the correct documents, and their ability to convey information accurately. This dependency often results in patients not fully comprehending the study details, their rights, and responsibilities, leading to reduced motivation to complete medical procedures or consent to additional testing.

The process of managing patient data consent can be made more efficient, transparent, and secure by using blockchain technology. Blockchain technology, being decentralized and impervious to tampering, offers a platform for creating a trustless environment, lowering the possibility of unwanted access, and giving patients more control over their data.

Smart contracts within a blockchain-based system provide a transformative solution for addressing these issues by ensuring secure and efficient management of consent documents through permissioned ledgers. The use of smart contracts automates and enforces compliant patient enrollment status Real-time, high-level overviews of consent progress are available, and new or updated consent forms can be seamlessly distributed to patients through the smart contracts. This ensures that patients always sign the correct version of the informed consent. Additionally, patients consent records are stored in a searchable, immutable database, providing secure and reliable time-stamped audit trails that facilitate oversight by ethics and regulatory bodies.

This analysis and design exploration will delve into the implementation of smart contracts within the consent management system, focusing on their role in enhancing the efficiency, security, and compliance of the process. By examining a blockchain-based consent platform, we will identify key design principles, system architecture, and potential benefits and challenges of integrating smart contracts into consent management for accessing health data. This study aims to provide a comprehensive understanding of how blockchain technology can revolutionize the consent management process, ultimately contributing to more effective and efficient health data management.

5.3.3.2 General Data Protection Regulation (GDPR)

Consent management is essential for ensuring that patients' wishes regarding their health data are respected and compliant with the General Data Protection Regulation (GDPR). The healthcare sector, like many others, has been bound by GDPR since it came into force in May 2018. Central to GDPR are the enhanced rights it grants individuals over their personal data. For patients, this translates to greater control over their health information. Key rights under GDPR include the Right to Access and the Right to be Forgotten. The Right to Access allows

patients to obtain copies of their personal data held by healthcare providers, while the Right to be Forgotten enables them to request the deletion of their data under certain conditions.
These rights highlight the importance of effective consent management systems that allow patients to easily exercise their GDPR rights. Robust consent management not only ensures GDPR compliance but also demonstrates a healthcare provider's commitment to upholding patient data privacy. This commitment is crucial for building and maintaining patient trust in an increasingly data-driven world. GDPR distinguishes between two types of data handlers: data controllers and data processors. In healthcare, a data controller can be the patient who determines the purpose and means of processing their personal data. The data processor, such as a hospital, processes personal data on behalf of the controller. Data processing includes storing, collecting, erasing, or organizing data.
GDPR has significantly impacted the healthcare industry by enforcing strict regulations on the collection, processing, and security of personal health data. It requires healthcare institutions to obtain patient consent for using their data, ensuring that collected data is used solely for specified purposes. GDPR also emphasizes that health data ownership and control should remain with the patients. Additionally, GDPR mandates that data processors, such as organizations that store and collect personal data, must report data breaches within a specific timeframe and notify the affected data controllers.

Table 2 provides a summary of key GDPR requirements relevant to the created scenarios, which must be adhered to when implementing Blockchain in healthcare. These requirements outline the specific GDPR regulations that healthcare organizations need to consider during the deployment and use of Blockchain technology. Simply put, the table highlights essential GDPR articles designed to protect sensitive patient data.

GDPR Article	Description	Impact on Healthcare
Art. 6	**Lawfulness of Processing**	Health data can only be processed if there is a justified purpose. Healthcare institutions must meet one of the six conditions for lawful data processing.
Art. 7	**Conditions for Consent**	Consent must be obtained from the patient before any data retrieval, and patients must have the option to withdraw their consent at any time.
Art. 17	**Right to Erasure/Right to be Forgotten**	Patients have the right to request the deletion of all their health data from the data processor.

Art. 32	**Security of Processing**	Personal data must be encrypted, pseudonymized, or anonymized during processing to ensure data security.

Table 2: Overview of GDPR Regulations Applicable to Healthcare

5.3.3.3 Enhanced Medical Data Management System

Enhanced Medical Data Management integrates Role-Based Access Control (RBAC) and a Consent Management System within a blockchain framework to ensure the secure and efficient handling of patient information.

RBAC is employed to assign specific roles and permissions, authorities based on users' identities and responsibilities. This approach restricts access to patient data to verified healthcare professionals while empowering patients to control the sharing of their personal information through consent mechanisms. This dual-layered strategy not only enhances data privacy but also enforces strict access controls, mitigating the risk of unauthorized data breaches.

Simultaneously, the Consent Management System utilizes smart contracts deployed on the blockchain. These contracts autonomously execute predefined rules embedded within their code to regulate data access permissions. By automating the enforcement of consent terms, such as who can access which data under what conditions. Smart contracts streamline administrative workflows and ensure compliance with regulatory requirements. This automated approach enhances transparency and accountability in data management practices, safeguarding patient confidentiality and integrity.

By leveraging blockchain technology's decentralized and immutable nature, Enhanced Medical Data Management strengthens data security by eliminating single points of failure and providing a tamper-resistant audit trail. This integration of RBAC and smart contract-driven consent management not only enhances operational efficiency within healthcare organizations but also reinforces patient trust through robust data protection measures aligned with global privacy standards.

Role-Based Access Control (RBAC)

The RBAC system developed for Enhanced Medical Data Management system serves as a foundational component ensuring secure and compliant handling of sensitive healthcare information. This RBAC implementation is designed to manage user roles and permissions effectively within blockchain-based platform, aiming to safeguard patient privacy while enabling authorized access as per regulatory requirements.
In system, distinct roles are defined to reflect the diverse stakeholders involved in healthcare operations, including patients, doctors, insurance companies, healthcare institutions, and administrators. Each role is equipped with specific permissions tailored to their responsibilities

and needs. For instance, verified doctors are granted access to patient data APIs, allowing them to retrieve medical records, update treatment plans, and conduct necessary diagnostic procedures securely.

RBAC policies enforce stringent access controls to prevent unauthorized access attempts. Only authenticated users with appropriate roles and credentials can interact with designated APIs and perform authorized actions within the blockchain network. This approach ensures that sensitive patient information remains accessible only to authorized personnel, thus mitigating the risks associated with data breaches and unauthorized disclosures.

By leveraging RBAC within the system, process prioritize data security and regulatory compliance (such as GDPR) while facilitating efficient healthcare operations. This structured access management framework not only enhances operational transparency and accountability but also fosters patient trust by safeguarding their confidential health information throughout its lifecycle within our blockchain-enabled patient data management solution.

RBAC Architecture

In the Spring Boot-based healthcare blockchain application, a Role-Based Access Control (RBAC) system has been implemented using Spring Security and JWT (JSON Web Token) authentication to enforce rigorous security measures for patient data. This approach begins with a comprehensive authentication flow where users, such as doctors or administrators, authenticate themselves using their credentials upon accessing the application. Once authenticated, a JWT token is generated and issued to the user, containing encoded information about their identity and associated roles within the system.

Subsequently, for each API request made by the user, this JWT token is sent in the authorization header. Spring Security intercepts these requests and performs rigorous verification to validate the token's authenticity and determine its validity period. This verification process is pivotal in securing patient data, as it ensures that only authenticated and authorized users can access the system's resources.

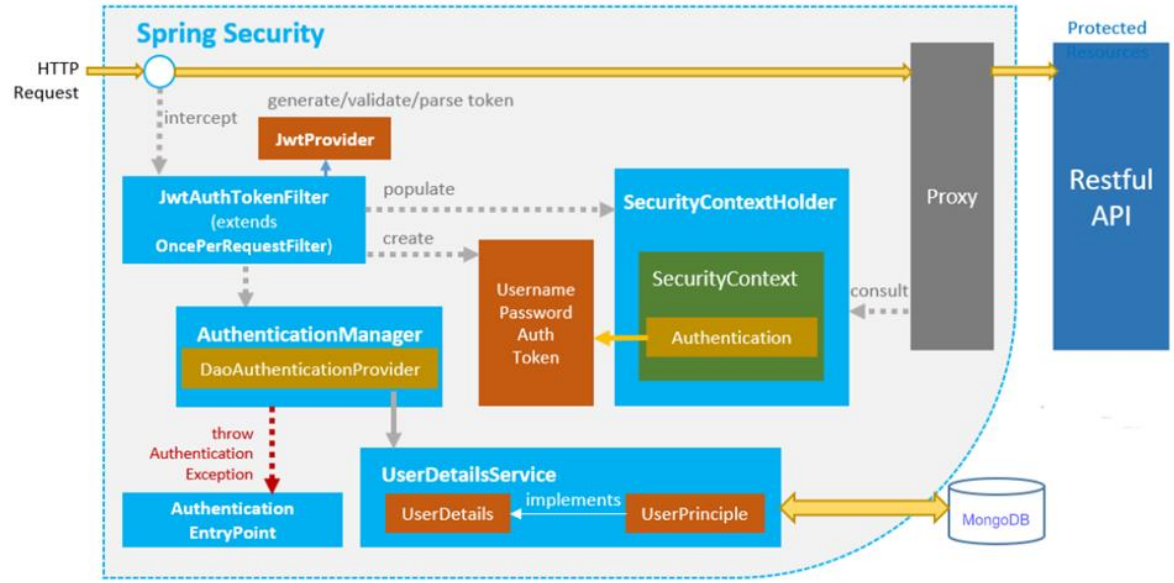


Figure 21: Architecture of Spring Security for RBAC

RBAC forms the backbone of access management strategy. Users are assigned specific roles: doctors, administrators, patients during registration and through administrative settings. These roles are meticulously mapped to permissions that dictate the actions a user can perform within the application. For instance, a doctor may have permissions to retrieve patient records or update treatment plans, while an admin has broader access to manage user roles and system configurations like block mining.

When integrating RBAC with a consent management system in blockchain application, stringent access controls are enforced to ensure that only verified doctors and authorized parties within the blockchain network can participate in the consent management module. For instance, only verified healthcare providers, authenticated through JWT tokens and verified against their assigned roles (e.g., ROLE_DOCTOR), are granted access to the consent management system. Unauthorized attempts to access the consent management module by users who are not verified or lack appropriate roles are automatically rejected by the RBAC system. This ensures that only authorized personnel, who have undergone the necessary verification processes and meet the defined criteria, can interact with sensitive functionalities such as managing patient consents within the blockchain application.

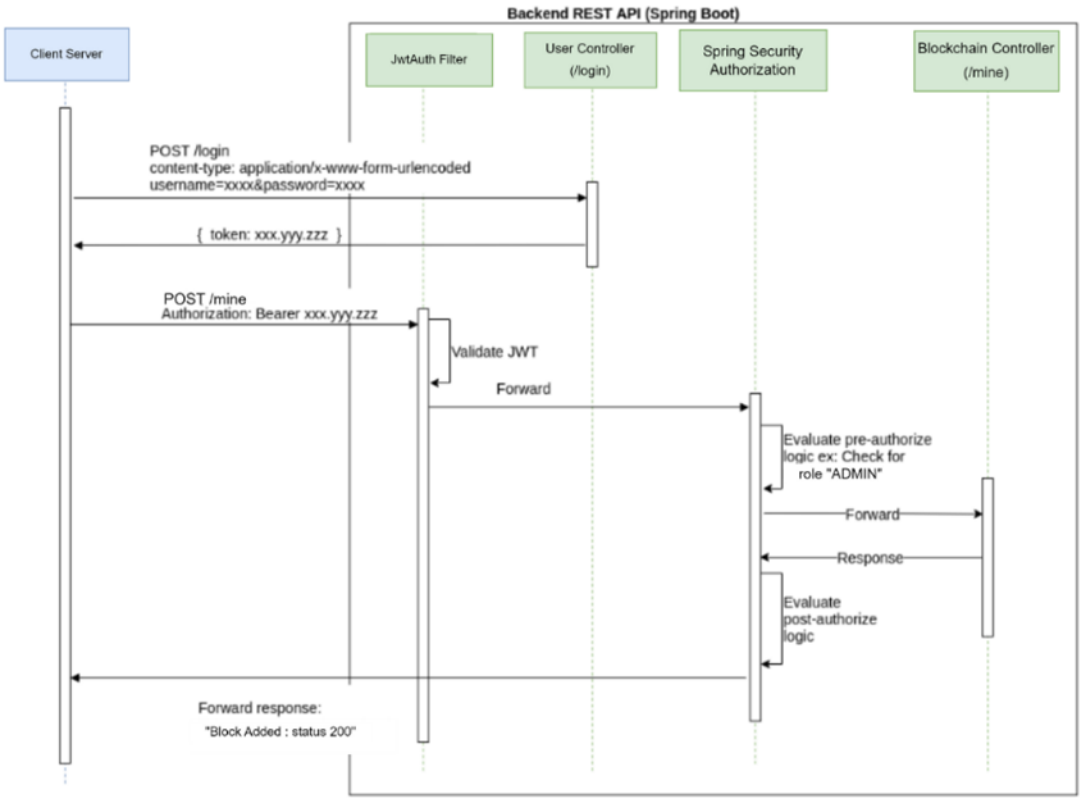


Figure 22: RBAC flow with Spring Security and JWT

API access within application is meticulously controlled using method-level security annotations provided by Spring Security. These annotations, such as `@PreAuthorize("hasAuthority('ROLE_DOCTOR')")`, are applied to API endpoints to restrict access based on the user's roles and permissions. This ensures that sensitive operations related to patient data, such as retrieving medical histories or modifying treatment protocols, are accessible only to authorized personnel. Unauthorized attempts to access these endpoints are systematically blocked, thus fortifying the application against data breaches and unauthorized data modifications.

Design includes robust exception handling mechanisms provided by Spring Security. These mechanisms manage authentication failures, expired tokens, and unauthorized access attempts gracefully, ensuring that appropriate error responses are returned by maintaining accountability and facilitating compliance with healthcare regulatory standards such as GDPR.

Overall integration of Spring Security and JWT authentication within the Spring Boot blockchain application not only enhances security but also ensures compliance with healthcare data protection regulations.

Consent Management System (CMS)

Consent Management System (CMS) represents an innovative solution designed specifically to streamline patient data management in healthcare blockchain network while ensuring robust protection and compliance with regulatory standards. Utilizing a series of smart contracts deployed on a blockchain framework, this CMS addresses the critical need for secure and transparent management of patient consents in healthcare.

At the core of this CMS are smart contracts, which serve as digital protocols encoded with predefined rules and conditions. These contracts automate the process of granting, initializing, revoking, and securely storing patient consents, leveraging blockchain's inherent security and immutability. This approach not only enhances the efficiency of consent management but also ensures that patient data remains confidential and accessible only to authorized healthcare providers.

This system can achieve greater transparency and accountability in how patient data is accessed and utilized. Patients gain greater control over their data, with the ability to grant or withdraw consent through a user-friendly interface that interacts seamlessly with our blockchain-based system.

1. Smart Contracts

Smart contracts, first conceptualized by Nick Szabo in 1997, have become a cornerstone of blockchain technology, particularly on platforms like Ethereum. These digital agreements operate autonomously as small programs stored on a blockchain, offering a decentralized alternative to traditional contracts. Unlike traditional contracts handled by intermediaries, smart contracts execute based on predefined conditions coded into their immutable structure.

Smart contracts have revolutionized blockchain technology by enabling automated, trustless agreements between parties. Initially introduced on Ethereum and later adapted by other cryptocurrencies, smart contracts allow developers to create decentralized applications that execute predefined actions when specific conditions are met. These contracts are stored on the blockchain, ensuring immutability and transparency, once deployed, their code cannot be altered without creating a visible record on the blockchain. Furthermore, their execution is validated by multiple nodes in the network, reducing the risk of fraud or manipulation by any single party.

The beauty of smart contracts lies in their ability to streamline transactions without relying on intermediaries. This autonomy not only reduces costs associated with third-party intermediaries but also enhances security and accuracy, as transactions are validated by the distributed network rather than a single entity. As a result, smart contracts are increasingly seen as a cornerstone of efficient, secure, and cost-effective digital transactions across various industries.

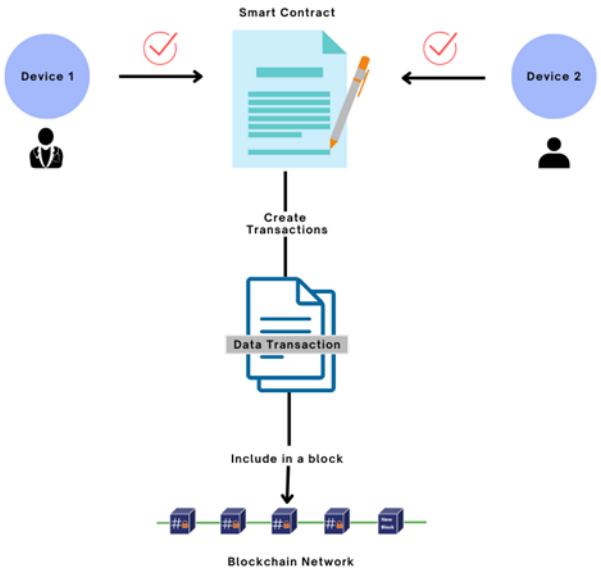


Figure 23: Smart Contract transactions with Blockchain Network

Smart contracts, often written in languages like Solidity, are tailored for specific blockchain platforms like Ethereum, where Solidity is widely supported by developers for its suitability in coding smart contracts. These contracts utilize storage variables such as mappings, arrays, and structs to store data directly on the Ethereum blockchain's state. This approach ensures that all data storage and retrieval processes occur within the decentralized Ethereum network, enhancing security and eliminating the need for external databases. By leveraging blockchain's decentralization and immutability, smart contracts maintain persistent data records that are secure and resistant to tampering, making them ideal for applications requiring long-term data storage and reliable access. This inherent persistence underscores blockchain's capability to provide a robust and trustworthy framework for managing data-intensive applications in various domains, including healthcare and financial sectors.

1. ***Deploying Smart Contracts***

The process of writing and deploying smart contracts on a blockchain network begins with selecting the appropriate platform, such as Ethereum, and choosing a suitable programming language like Solidity. Developers then proceed to write the smart contract code, defining its functionality, variables, and operational logic. This coding phase is crucial as it determines how the contract will autonomously execute tasks once deployed on the blockchain.

Following coding, rigorous testing and debugging ensue to identify and rectify any potential vulnerabilities or errors in the smart contract code. Test environments like Ethereum's testnets (e.g., Rinkeby, Ropsten, Ganache) allow developers to simulate blockchain conditions without using real cryptocurrency, ensuring that the contract behaves as expected under various scenarios.

Once testing is complete and the code is error-free, the next step is to compile the smart contract code into bytecode and generate ABI. (Task 1 figure 2) This compilation process translates the high-level code into machine-readable instructions that the Ethereum Virtual Machine (EVM) can execute. Deployment involves uploading the compiled bytecode of the smart contract onto the chosen blockchain network. . (Task 2 figure 2) Tools like Remix IDE or Truffle, along with web interfaces provided by blockchain platforms, facilitate this deployment process. During deployment, developers specify parameters such as gas limits (transaction fees) and initial configurations for the contract.

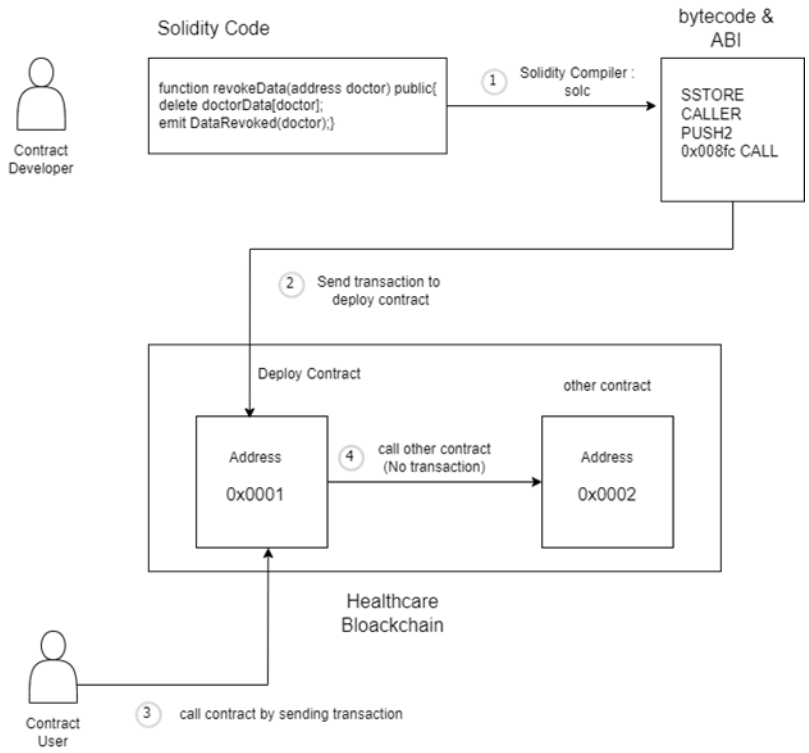


Figure 23: Flow of Deploying Smart Contracts into Bloackchain Network

After deployment, users and other contracts can interact with the deployed smart contract through transactions. . (Task 3 figure 2) External parties invoke functions defined within the smart contract, triggering specific actions or changes in its state. These interactions are

recorded on the blockchain, ensuring transparency and immutability of all executed transactions.

It's important to note that calling another smart contract within the blockchain network does not initiate a new transaction. . (Task 4 figure 2) However, when an external party interacts with a smart contract, such as invoking its functions or sending transactions and it constitutes a transaction on the blockchain. Post-deployment, ongoing monitoring, maintenance, and potential updates to the contract's code are essential. Regular audits and security checks help maintain the integrity and security of the deployed smart contract throughout its lifecycle. This comprehensive approach ensures that smart contracts effectively automate and secure digital agreements and processes within the decentralized blockchain ecosystem.

CMS Design

The design involves three primary participants: a doctor, a patient, and a data host which responsible for storing patient health data. Communication occurs exclusively through Blockchain technology, facilitating direct interaction among the participants. Within the Blockchain network, multiple smart contracts are deployed, each serving specific roles and responsibilities.

The data host is strictly considered untrusted, serving solely as an off-chain storage solution due to the impracticality of storing extensive health data directly on the Blockchain. It is assumed that patients have already stored their health data with this off-chain storage solution. Blockchain technology is employed primarily for facilitating secure and transparent transactions and interactions between participants in the system. It serves as a decentralized ledger where smart contracts manage and enforce data access and consent protocols.

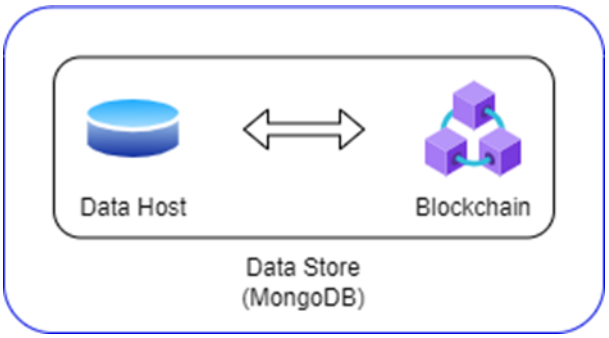


Figure 24: MongoDB Data Store

In this instance, we utilize a MongoDB database named "Data Store" to manage both blockchain and data host functionalities. (figure) However, for design clarity and efficiency, we treat blockchain and the data host as distinct entities. This separation ensures a clearer delineation of roles and responsibilities within our solution architecture.

Figure 13 illustrates the interaction between participants facilitated by smart contracts within the healthcare Blockchain network. This design framework ensures secure and efficient management of patient health data, leveraging Blockchain's decentralized architecture to enhance transparency and data integrity across all involved parties.

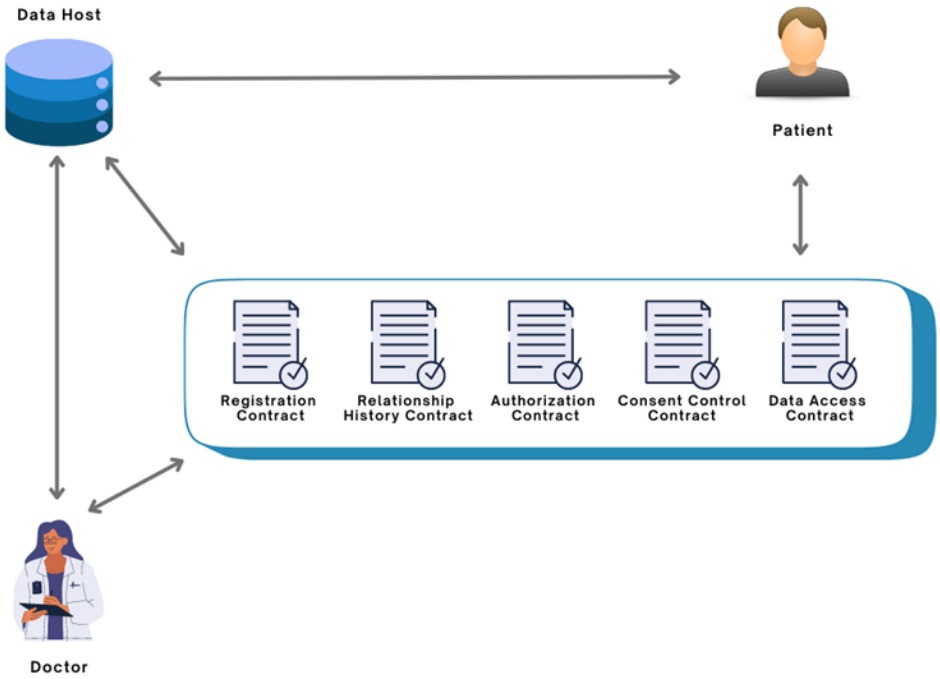


Figure 24: Overview of Consent Management Design

Next, consider the series of smart contracts utilized in the design and their respective purposes.

1. Registration Contract
   The Registration Contract is responsible for managing user registration within the Consent Management System. When a doctor requests patient data, this contract is the first to execute, verifying whether the doctor is registered. It also facilitates the creation of relationships between patients and doctors during the registration process. Furthermore, the Registration Contract validates and registers new nodes, ensuring their integration into the healthcare blockchain.

2. Relationship History Contract

   The Relationship History Contract is established during the registration process and is responsible for maintaining the relationship histories of users. It provides a detailed record of both previous and current healthcare relationships for patients. Crucially, the patient's consent is stored within this contract, capturing multiple consent stages rather than a simple true/false boolean status.

   The consent types include:
   - Granted: Data access request is accepted by the patient and consent is granted.
   - Requested: Initial status when a doctor requests access to patient data.
   - Authorized: Consent status after verifying that the doctor has permission to access data in a specific manner (e.g., READ, WRITE, UPDATE, DELETE).
   - Rejected: Patient declines the request for data access.
   - Revoked: Patient withdraws previously granted access permissions.

- Expired: The validity period of the consent has expired.
- Invalid: Doctor lacks the necessary permissions to access the data.

This contract ensures that consent management is thorough, transparent, and adaptable to various access scenarios within the healthcare blockchain network.

3. Authorization Contract
   The Authorization Contract defines the various permissions assigned to each participant, specifying the level of authorization a particular doctor has for a patient (e.g., READ, WRITE, UPDATE, DELETE). This contract stores the address of the Relationship History Contract and ensures that all data access requests are routed through it. Upon initialization, it updates the consent status in accordance with the specified authorization levels.

4. Consent Control Contract

   This smart contract activates when a request is initialized and subsequently authorized, allowing a patient to accept and sign the consent request. It also enables the data host to verify whether the patient has given consent. This contract empowers the patient with control over their consent.

5. Data Access Contract

   The Data Access Contract is responsible for retrieving data from the data host and providing it to the doctor who requested it, utilizing a query link after the patient grants consent. It acts as a pointer to a specific dataset in off-chain storage, containing the necessary information to locate the data. The contract includes a query link for the encrypted data, and the address of this main contract is stored within the Relationship History Contract.

Consent Management Flow

Two scenarios demonstrate consent management in our CMS design. The first scenario illustrates a doctor requesting access to a patient's health data. The second scenario depicts a patient revoking a doctor's access after initially granting consent to the requested dataset.

5.3.3.3.1.1 *Scenario 01 – Requesting to Access Data*

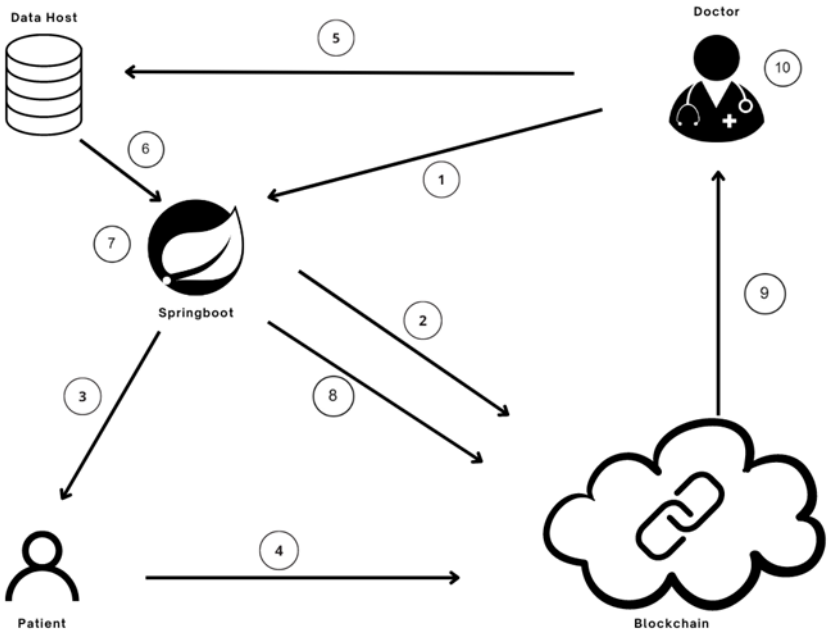


Figure 25: Consent Management Flow

1. A doctor initiates a request to access a patient's health data from the springboot app.
2. The system verifies if the doctor is registered:
   - If not registered, the doctor must undergo registration to CMS. This process involves using multiple smart contracts to establish a relationship between the doctor and the patient, thereby creating a Relationship History Contract.
   - If registered, the request proceeds to the authorization contract to confirm the doctor's permission level and the request is added as a transaction.(figure)
3. The patient receives notification of the access request with request information.
4. Then Patient provides consent using a consent control smart contract on the Blockchain and we assume that the patient signs the consent through a smart contract in Blockchain.
5. Upon the patient's consent, the doctor is notified that access has been granted and initiates retrieval of the data from the data host.
6. The data host securely transmits the requested encrypted dataset to the springboot app.
7. The system decrypts the data from the data host and encrypts it again using the doctor’s public key to ensure that only the authorized doctor can access it with their private key. A query link with a copy of the encrypted health data is then generated.
8. A Blockchain transaction containing the query link of the newly encrypted data is sent to the Data Access Contract. This contract store query link as a pointer in blockchain with relationship data.
9. The doctor retrieves the query link from the Data Access Contract within the Blockchain.
10. Using their private key, the doctor accesses the data through the query link securely stored within the Blockchain.

Adding a Request

In the process shows in Figure 16, the registration of a doctor occurs as described in step 2. The Authorization Contract maintains the address of the Relationship History Contract and validates whether the doctor is authorized to request patient data.

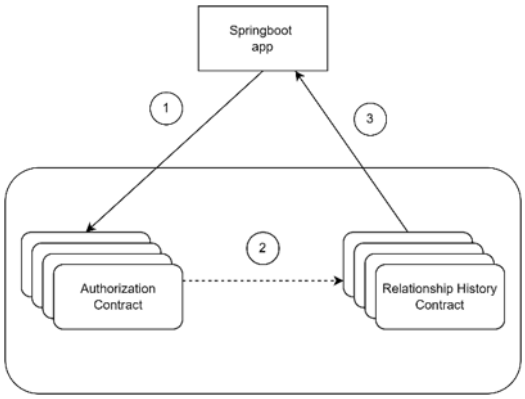


Figure 25: Adding a Request

1. When initiating a request, the system first contacts the Authorization Contract to verify the doctor's access permissions for the patient's data.
2. Upon confirmation of permission by the Authorization Contract, the Relationship History Contract updates the consent status to "permissioned." This action registers the request by storing it as a transaction on the Blockchain.
3. Following this, a message is relayed to the system to request patient consent and notify them accordingly through the system.

   This process ensures that only authorized doctors can request access to patient data, maintaining robust security and compliance within the blockchain-enabled healthcare system.

5.3.3.3.1.2 ***Scenario 02 - Revoking Given Permission***

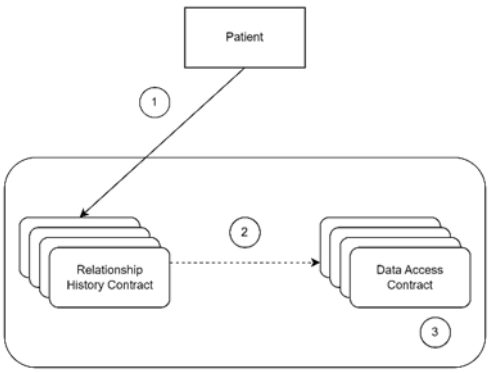


Figure 26: Revoking Given Permission

1. The patient initiates the revocation of access granted to the doctor through the CMS system, utilizing the relationship history contract.
2. The consent status in the relationship history contract is updated accordingly, triggering an action in the data access contract.
3. Subsequently, the data access contract identifies the specific location of the data and determines which query link needs removal. After confirming the details, the data access contract successfully deletes the query link, ensuring that access to the data is revoked.

5.3.3.4 Summary

In conclusion, Module 03 establishes a robust framework for streamlining patient data consent management processes in the healthcare sector. Blockchain's decentralized and transparent nature not only enhances security but also empowers patients by placing them at the centre of control over their sensitive health information. This module lays the groundwork for a consent management system that aligns with ethical principles, regulatory requirements, and the evolving needs of healthcare stakeholders.

5.3.4 - Module 04 - Facilitating Anonymous Data Provision for Research and Business Analytics

5.3.4.1- Introduction

Module 04 presents a novel approach to facilitate anonymous data provision for research and business analytics using Attribute-Based Encryption (ABE) integrated with a Java-based blockchain. The method ensures secure, anonymous, and fine-grained access control to sensitive data, making it suitable for diverse applications in healthcare, finance, and other sectors. Encrypted data is stored on the blockchain, and decryption keys are issued to authorized parties, allowing secure access through a query link. Traditional encryption methods typically rely on the use of public and private keys for encrypting and decrypting data. While effective, these methods can be cumbersome when dealing with complex access control requirements. ABE allows for more flexible and dynamic access control by tying decryption capabilities to a set of attributes rather than specific keys. This means that data can be encrypted with a policy specifying which attributes are required for decryption, such as "Doctor" or "Researcher" with certain credentials. Users possessing the necessary attributes in their keys can decrypt the data, providing a powerful way to enforce access control policies directly through the encryption mechanism.
Why ABE unique for our solution:

Fine-Grained Access Control
- ABE allows data owners to specify precise access control policies, ensuring that only authorized users with specific attributes can access sensitive data.

Scalability
- ABE is scalable in scenarios where multiple users need access to different parts of the data based on their roles or attributes, eliminating the need for multiple key distributions.

Privacy Preservation

-By allowing encrypted data to be shared without revealing the underlying information to unauthorized users, ABE enhances privacy and ensures compliance with data protection regulations.

Reduced Key Management Complexity

-Since access control policies are embedded in the encryption process, the complexity of key management is reduced, making it easier to maintain secure systems

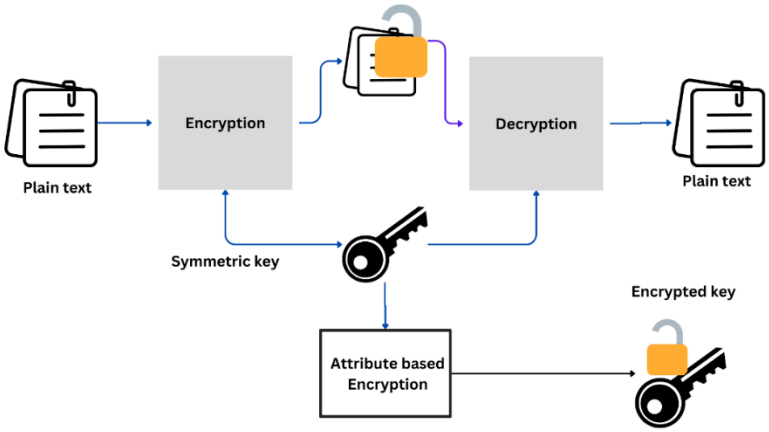


Figure : CP-ABE mechanism

5.3.4.1 Attribute Based Encryption (ABE) Workflow

Explanation of Workflow

- Define Access Policy

The data owner specifies an access policy based on attributes such as roles and departments (e.g., "Role: Doctor AND Department: Cardiology").

- Encrypt Data using CP-ABE:

The data is encrypted using CP-ABE, embedding the access policy directly into the ciphertext. This ensures that only users with matching attributes can decrypt the data.

- Store Encrypted Data on Blockchain:

The encrypted data (ciphertext) is stored on a Java-based blockchain, ensuring immutability and security.

- Authorized User Request:

An authorized user requests access to the encrypted data.

- Issue Private Key:

The attribute authority issues a private key to the user based on their attributes. This key is generated securely and provided to the user.

- Decrypt Data using Private Key:

The authorized user uses the private key to decrypt the data, allowing them to access the information if their attributes satisfy the access policy embedded in the ciphertext.

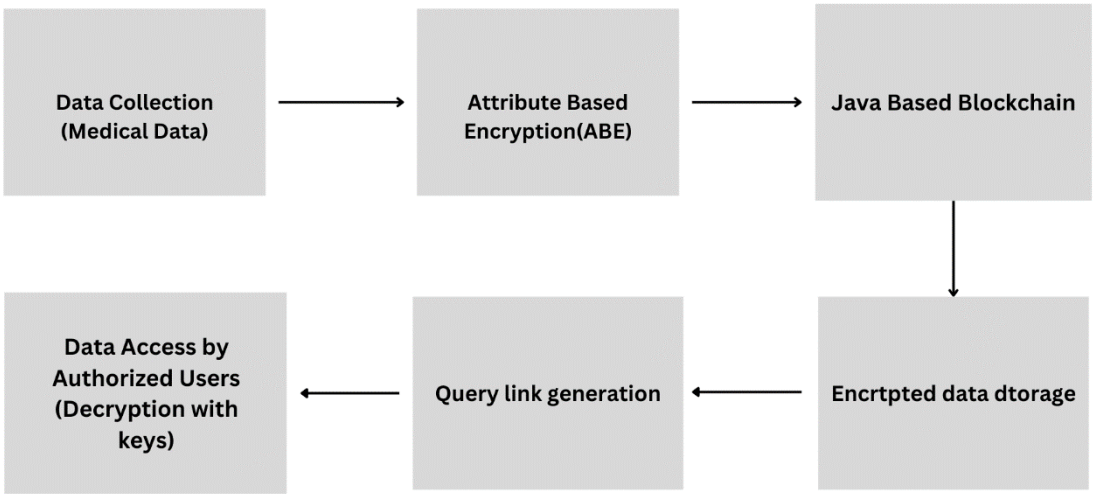
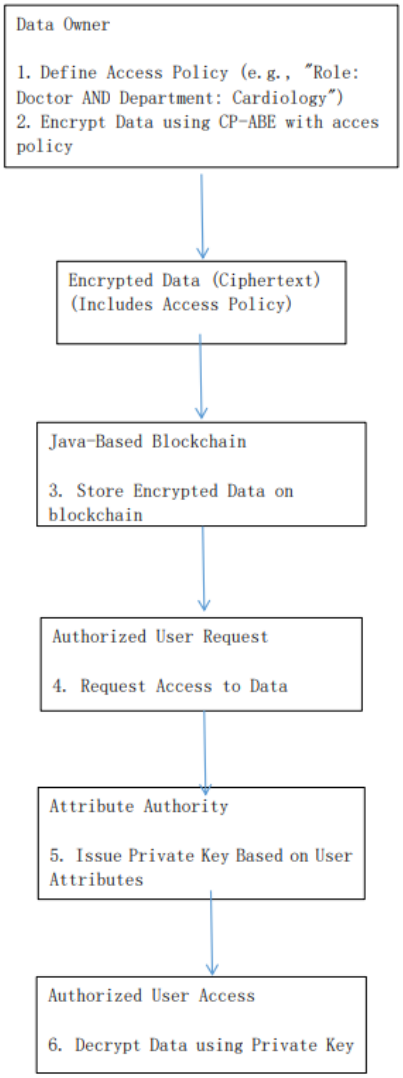


Figure: Architecture diagram with ABE

5.3.4.2 Detailed Steps in CP-ABE Workflow

- Access Policy Definition:
  The data owner defines an access policy that specifies which attributes are required to decrypt the data.
- Encryption:
  The data is encrypted using CP-ABE, producing a ciphertext that incorporates the access policy.
- Storage on Blockchain:
  The encrypted data is stored on the blockchain. Each block contains metadata and the hash of the previous block, ensuring data integrity and immutability.
- Access Request
  An authorized user requests access to the encrypted data. This request can be made through a query link generated by the system.
- Private Key Issuance:
  The attribute authority verifies the user's attributes and issues a private key if the user's attributes match the access policy.
- Decryption:
  The user uses the private key to decrypt the data. If the user's attributes satisfy the access policy, the data is successfully decrypted and accessed.



The diagram and workflow provide a clear overview of how Ciphertext-Policy Attribute-Based Encryption (CP-ABE) is used in conjunction with a Java-based blockchain to facilitate secure and anonymous data provision. This approach ensures that only authorized users with the correct attributes can access sensitive data, enhancing both security and privacy.

5.3.4.3 Security Analysis

To ensure the robustness of the proposed system, a comprehensive security analysis was conducted. This analysis aimed to identify potential vulnerabilities and address them to enhance the overall security of the system.

Data Integrity:

- Mechanism: The blockchain's immutable ledger ensures that once data is stored, it cannot be altered or tampered with.

- Analysis: The use of cryptographic hashing and linking of blocks provides strong guarantees of data integrity. Any attempt to alter the data would require modifying all subsequent blocks, which is computationally infeasible.

Access Control:

- Mechanism: CP-ABE enforces fine-grained access control based on user attributes and access policies.
- Analysis: The dynamic attribute management and real-time access policy adjustment ensure that only authorized users with the correct attributes can decrypt and access the data. Unauthorized users cannot decrypt the data, even if they have access to the encrypted ciphertext.

Confidentiality:

- Mechanism: Data is encrypted using CP-ABE before being stored on the blockchain.
- Analysis: The encryption ensures that sensitive medical data remains confidential and cannot be accessed by unauthorized parties. The decryption keys are issued based on attributes, adding an additional layer of security.

Potential Vulnerabilities:

- Key Management: The complexity of managing encryption keys and attributes can introduce vulnerabilities if not handled securely.
    - Mitigation: Implement secure key management practices, including periodic key rotation and secure storage of keys.
- Blockchain Security: While blockchain provides strong security guarantees, potential vulnerabilities such as 51% attacks need to be considered.
    - Mitigation: Utilize robust consensus mechanisms and ensure a distributed network of nodes to mitigate the risk of such attacks.

5.3.4.4 Summary

This chapter discussed the high-level design of the overall blockchain model. Furthermore, it dived deeper into the concepts and aspects of each underlying module that contributed towards the proper implementation of the novel model. The next chapter will discuss primarily the implementation of the blockchain model developed to securely store patient data and to streamline the patient data consent management process.

CHAPTER 06 : IMPLEMENTATION

6.1 - Introduction

This chapter discusses the steps taken to implement the novel model. It also highlights how the implementation of each module has contributed towards the functionality of the entire proposed system that could be used to securely store patient data.

6.2 - Implementation of individual modules.

6.2.1 - Module 01: Develop a Blockchain Model to Securely Store Medical Data and Propose a Novel Algorithm (Cross-Hash Validator Algorithm) to Enhance the Security of the Stored Data

6.2.1.1 Development of the blockchain

The blockchain model was developed using Java programming language and the Spring Boot framework by following the MVC (Model-View-Controller) architecture. IntelliJ Idea was used as the go to development environment to develop the backend of the blockchain platform while ReactJS framework was used to develop the frontend view of the web application that is accessible to users.

When it comes to the development of the blockchain platform using Spring Boot and Java technologies by following the MVC architecture, several model classes, controller classes, services classes was need to be written to achieve the ….

Let's look at the directory structure of the project for the backend integration of the blockchain platform.

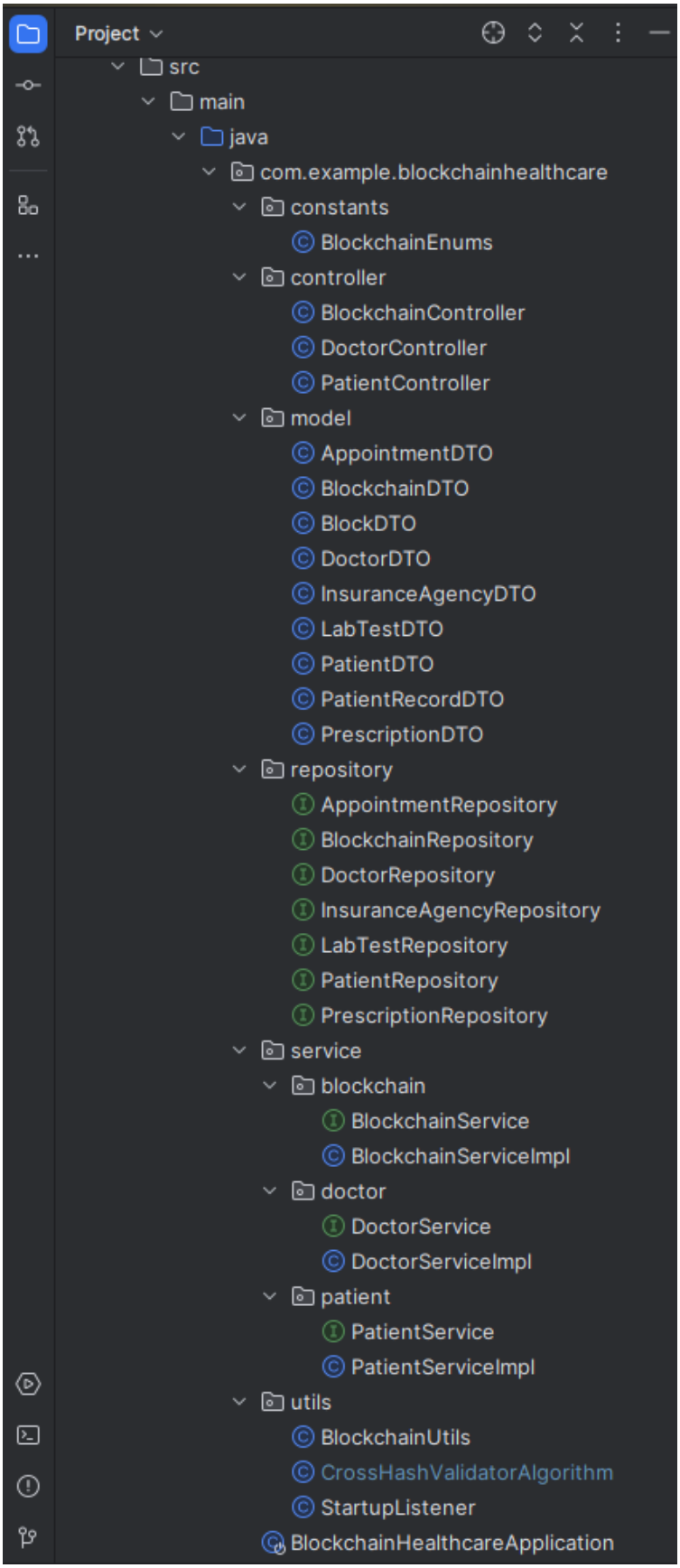


Figure 20: Directory structure of the project for the backend integration of the blockchain platform.

Directory Structure of the project for the backend integration of the blockchain platform.

1. **constants**
    - **BlockchainEnums**
        - Contains enums used throughout the application for defining constants related to blockchain entities and actions.
2. **controller**
    - **BlockchainController**
        - Manages requests related to blockchain operations.
    - **DoctorController**
        - Handles HTTP requests for doctor-related operations.
    - **PatientController**
        - Manages HTTP requests for patient-related operations.
3. **model**
    - Contains Data Transfer Objects (DTOs) representing the data structures used in the application.
    - **AppointmentDTO**
        - Represents appointment-related data.
    - **BlockDTO**
        - Represents block-related data in the blockchain.
    - **DoctorDTO**
        - Represents doctor-related data.
    - **InsuranceAgencyDTO**
        - Represents insurance agency-related data.
    - **LabTestDTO**
        - Represents lab test-related data.
    - **PatientDTO**
        - Represents patient-related data.
    - **PrescriptionDTO**
        - Represents prescription-related data.
4. **repository**
    - Contains interfaces for database operations using Spring Data MongoDB.
    - **AppointmentRepository**
        - Interface for CRUD operations on appointments.
    - **BlockchainRepository**
        - Interface for CRUD operations on blockchain data.
    - **DoctorRepository**
        - Interface for CRUD operations on doctors.
    - **InsuranceAgencyRepository**
        - Interface for CRUD operations on insurance agencies.
    - **LabTestRepository**
        - Interface for CRUD operations on lab tests.
    - **PatientRepository**
        - Interface for CRUD operations on patients.
    - **PrescriptionRepository**
        - Interface for CRUD operations on prescriptions.
5. **service**
    - Contains service classes that implement business logic.
    - **blockchain**

        - **BlockchainService**
            - Interface defining methods for blockchain-related operations.
        - **BlockchainServiceImpl**
            - Implements BlockchainService, handling blockchain operations.
    - **doctor**
        - **DoctorService**
            - Interface defining methods for doctor-related operations.
        - **DoctorServiceImpl**
            - Implements DoctorService, handling doctor-related operations.
    - **patient**
        - **PatientService**
            - Interface defining methods for patient-related operations.
        - **PatientServiceImpl**
            - Implements PatientService, handling patient-related operations.
6. **utils**
    - Contains utility classes and methods supporting various functionalities.
    - **BlockchainUtils**
        - Utility methods for blockchain operations.
    - **CrossHashValidatorAlgorithm**
        - Implements the Cross-Hash Validator Algorithm for enhancing data security and integrity.
    - **StartupListener**
        - Executes certain tasks during the application startup.

Lets explore each of these classes in detail in the following section.

Constant Classes

BlockchainEnums: Enum constants for various blockchain-related definitions, such as action types and entity names.

```
package com.example.blockchainhealthcare.constants;

public class BlockchainEnums {
    public enum HTTPMethodType {
        POST("post"),
        PUT("put"),
        DELETE("delete");

        private final String action;

        HTTPMethodType(String action) {
            this.action = action;
        }

        @Override
        public String toString() {
            return action;
        }
    }

    public enum CollectionType {
        PATIENTS("patients"),
        DOCTORS("doctors");
        private final String collection_type;

        CollectionType(String collection_type) {
            this.collection_type = collection_type;
```

```
        }

        @Override
        public String toString() {
            return collection_type;
        }
    }
}
```

Controller Classes

BlockchainController: Manages blockchain-specific endpoints, handling requests for creating and managing blockchain records.

```
package com.example.blockchainhealthcare.controller;

import com.example.blockchainhealthcare.model.BlockDTO;
import
com.example.blockchainhealthcare.service.blockchain.BlockchainService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;

@RestController
@RequestMapping("/api/blockchain")
public class BlockchainController {

    @Autowired
    private BlockchainService blockchainService;

    @GetMapping("/full")
    public ArrayList<BlockDTO> getFullBlockchain() {
        return blockchainService.getAllBlocks();
    }

    @GetMapping("/validate")
    public String validateBlockchain(){
        return blockchainService.validateBlockchain();
    }
}
```

DoctorController: Manages endpoints related to doctor operations, such as creating, updating, and retrieving doctor data.

```
package com.example.blockchainhealthcare.controller;

import com.example.blockchainhealthcare.model.DoctorDTO;
import com.example.blockchainhealthcare.service.doctor.DoctorService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;

@RestController
@RequestMapping("/api/blockchain")
public class DoctorController {
```

```
    @Autowired
    private DoctorService doctorService;

    @GetMapping("/doctors")
    public ResponseEntity<?> getAllDoctors() {
        ArrayList<DoctorDTO> doctors = new
ArrayList<>(doctorService.getAllDoctors());
        return new ResponseEntity<>(doctors, HttpStatus.OK);
    }

    @GetMapping("/doctor/{id}")
    public ResponseEntity<?> getADoctor(@PathVariable("id") String id) {
        try {
            return new ResponseEntity<>(doctorService.getADoctor(id),
HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(e.getMessage(),
HttpStatus.NOT_FOUND);
        }
    }

    @PostMapping("/doctor")
    public ResponseEntity<?> createDoctor(@RequestBody DoctorDTO doctor) {
        try {
            doctorService.createDoctor(doctor);
            return new ResponseEntity<DoctorDTO>(doctor, HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(e.getMessage(),
HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }


    @PutMapping("/doctor/{id}")
    public ResponseEntity<?> updateADoctor(@PathVariable("id") String id,
@RequestBody DoctorDTO doctor) {
        try {
            doctorService.updateDoctor(id, doctor);
            return new ResponseEntity<>("Doctor updated successfully!",
HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(e.getMessage(),
HttpStatus.NOT_FOUND);
        }
    }

    @DeleteMapping("/doctor/{id}")
    public ResponseEntity<?> deleteADoctor(@PathVariable("id") String id) {
        try {
            doctorService.deleteDoctor(id);
            return new ResponseEntity<>("Doctor deleted successfully!",
HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(e.getMessage(),
HttpStatus.NOT_FOUND);
        }
    }
}
```

PatientController: Manages endpoints for patient-related operations, handling CRUD operations on patient data.

```
package com.example.blockchainhealthcare.controller;

import com.example.blockchainhealthcare.model.PatientDTO;
import com.example.blockchainhealthcare.service.patient.PatientService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;

@RestController
@RequestMapping("/api/blockchain")
public class PatientController {

    @Autowired
    private PatientService patientService;

    @GetMapping("/patients")
    public ResponseEntity<?> getAllPatients() {
        ArrayList<PatientDTO> patients = patientService.getAllPatients();
        return new ResponseEntity<>(patients, HttpStatus.OK);
    }

    @GetMapping("/patient/{id}")
    public ResponseEntity<?> getAPatient(@PathVariable("id") String id) {

        try {
            return new ResponseEntity<>(patientService.getAPatient(id),
HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(e.getMessage(),
HttpStatus.NOT_FOUND);
        }

    }

    @PostMapping("/patient")
    public ResponseEntity<?> createPatient(@RequestBody PatientDTO patient)
{
        try {
            patientService.createPatient(patient);
            return new ResponseEntity<PatientDTO>(patient, HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(e.getMessage(),
HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }


    @PutMapping("/patient/{id}")
    public ResponseEntity<?> updateAPatient(@PathVariable("id") String id,
@RequestBody PatientDTO patient) {

        try {
            patientService.updatePatient(id, patient);
            return new ResponseEntity<>("Patient updated succesfully!",
HttpStatus.OK);
        } catch (Exception e) {
```

```
            return new ResponseEntity<>(e.getMessage(),
HttpStatus.NOT_FOUND);
        }

    }

    @DeleteMapping("/patient/{id}")
    public ResponseEntity<?> deleteAPatient(@PathVariable("id") String id)
{
        try {
            patientService.deletePatient(id);
            return new ResponseEntity<>("Patient deleted successfully!",
HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(e.getMessage(),
HttpStatus.NOT_FOUND);
        }
    }
}
```

Model Classes

AppointmentDTO: Defines the data structure for Appointment Entity.

```
package com.example.blockchainhealthcare.model;

import jakarta.validation.constraints.NotNull;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.Date;

@Getter
@Setter
@AllArgsConstructor
@Document(collection = "appointments")
public class AppointmentDTO {
    @Id
    private String appointmentID;
    private int patientID;
    private int doctorID;
    private Date appointmentDateTime;
    private String purpose;

    @NotNull(message = "Date must not be null")
    private Date createdAt;

    @NotNull(message = "Date must not be null")
    private Date updatedAt;
}
```

BlockchainDTO: Defines the data structure for Blockchain Entity

```
package com.example.blockchainhealthcare.model;

import com.example.blockchainhealthcare.repository.BlockchainRepository;
import
com.example.blockchainhealthcare.service.blockchain.BlockchainService;
```

```
import com.example.blockchainhealthcare.utils.CrossHashValidatorAlgorithm;
import lombok.Getter;
import lombok.Setter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.util.ArrayList;

@Getter
@Setter
@Component
public class BlockchainDTO {
    private ArrayList<BlockDTO> chain = new ArrayList<>();

    @Autowired
    private BlockchainRepository blockchainRepo;
    private static BlockchainService blockchainService;

    public void setChain(ArrayList<BlockDTO> chain) {
        this.chain = chain;
    }
    public int generateBlockID() {
        if (chain.isEmpty()) {
            return 0;
        } else {
            return chain.size();
        }
    }

    public BlockDTO getPreviousBlock() {
        return chain.get(chain.size() - 1);
    }

    public String proofOfWork(int previousProof) {
        int newProof = 1;
        boolean checkProof = false;

        while (!checkProof) {
            String hashOperation =
CrossHashValidatorAlgorithm.hash(String.valueOf(newProof * newProof -
previousProof * previousProof));
            if (hashOperation.startsWith("0000")) {
                checkProof = true;
            } else {
                newProof++;
            }
        }
        return Integer.toString(newProof);
    }
}
```

BlockDTO: Defines the data structure for Block Entity.

```
package com.example.blockchainhealthcare.model;

import com.example.blockchainhealthcare.utils.CrossHashValidatorAlgorithm;
import lombok.Getter;
import lombok.Setter;
import org.springframework.data.mongodb.core.mapping.Document;
```

```
@Setter
@Getter
@Document(collection = "blockchain")
public class BlockDTO {

    private int block_id;
    private int proof;
    private String http_method;
    private String collection_name;
    private String document_id;
    private String data_hash;
    private String previousHash;
    private String hash;
    private long timestamp;

    public BlockDTO(int block_id, int proof,String http_method, String
collection_name, String document_id, String data_hash, String previousHash,
long timestamp) {
        this.block_id = block_id;
        this.proof = proof;
        this.http_method = http_method;
        this.collection_name = collection_name;
        this.document_id = document_id;
        this.data_hash = data_hash;
        this.previousHash = previousHash;
        this.timestamp = timestamp;
        generateBlockHash();
    }
    public void generateBlockHash(){
        this.setHash(CrossHashValidatorAlgorithm.hash(this.toString()));
    }

    @Override
    public String toString() {
        return "BlockDTO{" +
                "block_id=" + block_id +
                ", proof=" + proof +
                ", http_method='" + http_method + '\'' +
                ", collection_name='" + collection_name + '\'' +
                ", document_id='" + document_id + '\'' +
                ", data_hash='" + data_hash + '\'' +
                ", previousHash='" + previousHash + '\'' +
                ", timestamp=" + timestamp +
                '}';
    }
}
```

DoctorDTO: Defines the data structure for Doctor Entity.

```
package com.example.blockchainhealthcare.model;

import jakarta.validation.constraints.NotNull;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.Date;
```

```
@Getter
@Setter
@AllArgsConstructor
@ToString
@Document(collection = "doctors")
public class DoctorDTO {
    @Id
    private String doctorID;
    private String name;
    private Date dateOfBirth;
    private String gender;
    private String specialization;
    private String contactNumber;
    private String email;

    @NotNull(message = "Date must not be null")
    private Date createdAt;

    @NotNull(message = "Date must not be null")
    private Date updatedAt;
}
```

InsuranceAgencyDTO: Defines the data structure for InsuranceAgency Entity.

```
package com.example.blockchainhealthcare.model;

import jakarta.validation.constraints.NotNull;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.Date;

@Getter
@Setter
@AllArgsConstructor
@Document(collection = "insurance_agencies")
public class InsuranceAgencyDTO {
    @Id
    private String insuranceAgencyID;
    private String name;
    private String address;
    private String contactNumber;
    private String email;
    private String policyDetails;
    private String contactPerson;

    @NotNull(message = "Date must not be null")
    private Date createdAt;

    @NotNull(message = "Date must not be null")
    private Date updatedAt;
}
```

LabTestDTO: Defines the data structure for LabTest Entity.

```
package com.example.blockchainhealthcare.model;

import jakarta.validation.constraints.NotNull;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.Date;

@Getter
@Setter
@AllArgsConstructor
@Document(collection = "lab_tests")
public class LabTestDTO {
    @Id
    private String labTestID;
    private int patientID;
    private String testName;
    private Date testDate;
    private String result;
    private String doctorNotes;

    @NotNull(message = "Date must not be null")
    private Date createdAt;

    @NotNull(message = "Date must not be null")
    private Date updatedAt;
}
```

PatientDTO: Defines the data structure for PatientDTO Entity.

```
package com.example.blockchainhealthcare.model;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.Date;

@Getter
@Setter
@AllArgsConstructor
@ToString
@Document(collection = "patients")
public class PatientDTO {
    @Id
    private String patientID;
    private String name;
    private Date dateOfBirth;
    private String gender;
    private String address;
    private String contactNumber;
    private String email;
```

```
    private String insuranceAgencyID;
    private Date createdAt;
    private Date updatedAt;
}
```

PrescriptionDTO: Defines the data structure for PrescriptionDTO Entity.

```
package com.example.blockchainhealthcare.model;

import jakarta.validation.constraints.NotNull;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.Date;

@Getter
@Setter
@AllArgsConstructor
@Document(collection = "prescriptions")
public class PrescriptionDTO {
    @Id
    private String prescriptionID;
    private int patientID;
    private int doctorID;
    private Date prescriptionDate;
    private String diagnosis;
    private String medication;

    @NotNull(message = "Date must not be null")
    private Date createdAt;

    @NotNull(message = "Date must not be null")
    private Date updatedAt;
}
```

Repository Classes
Contains interfaces for database operations using Spring Data MongoDB.
AppointmentRepository: Interface for CRUD operations on appointments.

```
package com.example.blockchainhealthcare.repository;

import com.example.blockchainhealthcare.model.AppointmentDTO;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface AppointmentRepository extends
MongoRepository<AppointmentDTO,String> {
}
```

BlockchainRepository: Interface for CRUD operations on blockchain data.

```
package com.example.blockchainhealthcare.repository;

import com.example.blockchainhealthcare.model.BlockDTO;
import org.springframework.data.mongodb.repository.MongoRepository;
```

```
public interface BlockchainRepository extends MongoRepository<BlockDTO,
Integer>{
}
```

DoctorRepository: Interface for CRUD operations on doctors.

```
package com.example.blockchainhealthcare.repository;

import com.example.blockchainhealthcare.model.DoctorDTO;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface DoctorRepository extends MongoRepository<DoctorDTO,String>
{
}
```

InsuranceAgencyRepository: Interface for CRUD operations on insurance agencies.

```
package com.example.blockchainhealthcare.repository;

import com.example.blockchainhealthcare.model.AppointmentDTO;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface InsuranceAgencyRepository extends
MongoRepository<AppointmentDTO,String> {
}
```

LabTestRepository: Interface for CRUD operations on lab tests.

```
package com.example.blockchainhealthcare.repository;

import com.example.blockchainhealthcare.model.LabTestDTO;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface LabTestRepository extends
MongoRepository<LabTestDTO,String> {
}
```

PatientRepository: Interface for CRUD operations on patients.

```
package com.example.blockchainhealthcare.repository;

import com.example.blockchainhealthcare.model.PatientDTO;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface PatientRepository extends
MongoRepository<PatientDTO,String> {
}
```

PrescriptionRepository: Interface for CRUD operations on prescriptions.

```
package com.example.blockchainhealthcare.repository;

import com.example.blockchainhealthcare.model.PrescriptionDTO;
```

```
import org.springframework.data.mongodb.repository.MongoRepository;

public interface PrescriptionRepository extends
MongoRepository<PrescriptionDTO,String> {
}
```

Service Classes
BlockchainService and BlockchainServiceImpl

```
package com.example.blockchainhealthcare.service.blockchain;

import com.example.blockchainhealthcare.model.BlockDTO;

import java.util.ArrayList;

public interface BlockchainService {
    ArrayList<BlockDTO> getAllBlocks();

    String validateBlockchain();
}
```

```
package com.example.blockchainhealthcare.service.blockchain;

import com.example.blockchainhealthcare.model.BlockDTO;
import com.example.blockchainhealthcare.model.DoctorDTO;
import com.example.blockchainhealthcare.model.PatientDTO;
import com.example.blockchainhealthcare.repository.BlockchainRepository;
import com.example.blockchainhealthcare.repository.DoctorRepository;
import com.example.blockchainhealthcare.repository.PatientRepository;
import com.example.blockchainhealthcare.utils.CrossHashValidatorAlgorithm;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.HashMap;

@Service
public class BlockchainServiceImpl implements BlockchainService {

    @Autowired
    private BlockchainRepository blockchainRepo;
    @Autowired
    private PatientRepository patientRepository;
    @Autowired
    private DoctorRepository doctorRepository;

    @Override
    public ArrayList<BlockDTO> getAllBlocks() {
        ArrayList<BlockDTO> blocks = new
ArrayList<>(blockchainRepo.findAll());
        if (!blocks.isEmpty()) {
            return blocks;
        } else {
            return new ArrayList<>();
        }
    }

    @Override
    public String validateBlockchain() {
        ArrayList<BlockDTO> blockchain = new
ArrayList<>(blockchainRepo.findAll());
```

```
        ArrayList<PatientDTO> patients = new
ArrayList<>(patientRepository.findAll());
        ArrayList<DoctorDTO> doctors = new
ArrayList<>(doctorRepository.findAll());

        HashMap<String, String> records = new HashMap<>();

        // Add patients to the hash map
        for (PatientDTO patient : patients) {
            records.put(patient.getPatientID(), patient.toString());
        }
        // Add doctors to the hash map
        for (DoctorDTO doctor : doctors) {
            records.put(doctor.getDoctorID(), doctor.toString());
        }

        return CrossHashValidatorAlgorithm.validator(blockchain, records);
    }
}
```

DoctorService and DoctorServiceImpl

```
package com.example.blockchainhealthcare.service.doctor;

import com.example.blockchainhealthcare.model.DoctorDTO;

import java.util.ArrayList;
import java.util.List;

public interface DoctorService {

    ArrayList<DoctorDTO> getAllDoctors();

    DoctorDTO getADoctor(String id) throws Exception;

    void createDoctor(DoctorDTO doctor) throws Exception;

    void updateDoctor(String id, DoctorDTO doctor) throws Exception;

    void deleteDoctor(String id) throws Exception;
}
```

```
package com.example.blockchainhealthcare.service.doctor;

import com.example.blockchainhealthcare.constants.BlockchainEnums;
import com.example.blockchainhealthcare.model.BlockDTO;
import com.example.blockchainhealthcare.model.DoctorDTO;
import com.example.blockchainhealthcare.repository.BlockchainRepository;
import com.example.blockchainhealthcare.repository.DoctorRepository;
import com.example.blockchainhealthcare.utils.BlockchainUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.Date;
import java.util.Optional;

@Service
public class DoctorServiceImpl implements DoctorService {
```

```
    @Autowired
    private DoctorRepository doctorRepo;
    @Autowired
    private BlockchainRepository blockchainRepo;
    @Autowired
    private BlockchainUtils blockchainUtils;
    @Override
    public ArrayList<DoctorDTO> getAllDoctors() {
        ArrayList<DoctorDTO> doctors = new
ArrayList<>(doctorRepo.findAll());
        if (!doctors.isEmpty()) {
            return doctors;
        } else {
            return new ArrayList<>();
        }
    }

    @Override
    public DoctorDTO getADoctor(String id) throws Exception {
        Optional<DoctorDTO> optionalDoctor = doctorRepo.findById(id);
        if (optionalDoctor.isEmpty()) {
            throw new Exception("Doctor not found!");
        } else {
            return optionalDoctor.get();
        }
    }

    @Override
    public void createDoctor(DoctorDTO doctor) throws Exception {
        try {
            // Creating a new doctor in the DB
            if (doctor.getName().isEmpty() ||
                    doctor.getDateOfBirth() == null ||
                    doctor.getSpecialization().isEmpty() ||
                    doctor.getContactNumber().isEmpty()) {
                throw new Exception("Please enter all the fields!");
            }
            doctor.setCreatedAt(new Date(System.currentTimeMillis()));
            doctor.setUpdatedAt(new Date(System.currentTimeMillis()));
            doctorRepo.save(doctor);

            // Adding a block to the blockchain on POST request
            BlockDTO block =
blockchainUtils.mineBlock(BlockchainEnums.HTTPMethodType.POST,
BlockchainEnums.CollectionType.DOCTORS, doctor.getDoctorID(),
doctor.toString());
            blockchainRepo.save(block);

        } catch (Exception e) {
            throw new Exception(e.getMessage());
        }
    }

    @Override
    public void updateDoctor(String id, DoctorDTO doctor) throws Exception
{
        // Updating a doctor in the DB
        Optional<DoctorDTO> doctorOptional = doctorRepo.findById(id);

        if (doctor.getName().isEmpty() ||
                doctor.getDateOfBirth() == null ||
```

```
                doctor.getSpecialization().isEmpty() ||
                doctor.getContactNumber().isEmpty()) {
            throw new Exception("Please enter all the fields!");
        }

        if (doctorOptional.isEmpty()) {
            throw new Exception("Doctor not found!");
        } else {
            DoctorDTO doctorToSave = doctorOptional.get();

            // Name
            doctorToSave.setName(doctor.getName());
            // Date of Birth
            doctorToSave.setDateOfBirth(doctor.getDateOfBirth());
            // Gender
            doctorToSave.setGender(doctor.getGender());
            // Specialization
            doctorToSave.setSpecialization(doctor.getSpecialization());
            // Contact Number
            doctorToSave.setContactNumber(doctor.getContactNumber());
            // Email
            doctorToSave.setEmail(doctor.getEmail());
            // Updated At
            doctorToSave.setUpdatedAt(new
Date(System.currentTimeMillis()));

            doctorRepo.save(doctorToSave);

            // Adding a block to the blockchain on PUT request
            BlockDTO block =
blockchainUtils.mineBlock(BlockchainEnums.HTTPMethodType.PUT,
BlockchainEnums.CollectionType.DOCTORS, id, doctorToSave.toString());
            blockchainRepo.save(block);

        }
    }

    @Override
    public void deleteDoctor(String id) throws Exception {
        Optional<DoctorDTO> doctorOptional = doctorRepo.findById(id);

        if (doctorOptional.isEmpty()) {
            throw new Exception("Doctor not found!");
        } else {
            doctorRepo.deleteById(id);
            // Updating the blockchain on DELETE request
            BlockDTO block =
blockchainUtils.mineBlock(BlockchainEnums.HTTPMethodType.DELETE,
BlockchainEnums.CollectionType.DOCTORS, id, null);
            blockchainRepo.save(block);
        }
    }
}
```

The implementation of the service class for other entities such as patients, insurance agencies, lab reports etc are similar to that implementation of the service classes of the doctor entity.

Utility Classes

BlockchainUtils: Used for the creation of blocks in the blockchain.

```
package com.example.blockchainhealthcare.utils;
```

```
import com.example.blockchainhealthcare.constants.BlockchainEnums;
import com.example.blockchainhealthcare.model.BlockDTO;
import com.example.blockchainhealthcare.model.BlockchainDTO;
import
com.example.blockchainhealthcare.service.blockchain.BlockchainService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class BlockchainUtils {
    @Autowired
    private BlockchainService blockchainService;
    @Autowired
    private BlockchainDTO blockchainDTO;

    public BlockDTO mineBlock(BlockchainEnums.HTTPMethodType http_method,
BlockchainEnums.CollectionType collection_name, String document_id, String
data) {
        blockchainDTO.setChain(blockchainService.getAllBlocks());
        int previousProof = blockchainDTO.getPreviousBlock().getBlock_id();
        int proof =
Integer.parseInt(blockchainDTO.proofOfWork(previousProof));
        String previousHash = blockchainDTO.getPreviousBlock().getHash();
        String data_hash = data != null ?
CrossHashValidatorAlgorithm.hash(data) : null;
        return new BlockDTO(
                blockchainDTO.generateBlockID(),
                proof,
                http_method.toString(),
                collection_name.toString(),
                document_id,
                data_hash,
                previousHash,
                System.currentTimeMillis());
    }

}
```

CrossHashValidator Algorithm: Discussed under Chapter 5 Analysis and Design section.
StartupListener: Contains methods to run specific tasks when the application starts, such as initializing data or performing setup operations.

```
package com.example.blockchainhealthcare.utils;

import com.example.blockchainhealthcare.model.BlockDTO;
import com.example.blockchainhealthcare.model.BlockchainDTO;
import com.example.blockchainhealthcare.repository.BlockchainRepository;
import
com.example.blockchainhealthcare.service.blockchain.BlockchainService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

@Component
public class StartupListener {
    @Autowired
    private BlockchainDTO blockchainDTO;
    @Autowired
    private BlockchainRepository blockchainRepo;
    @Autowired
```

```
    private BlockchainService blockchainService;

    @EventListener(ApplicationReadyEvent.class)
    public void onApplicationEvent() {
        blockchainDTO.setChain(blockchainService.getAllBlocks());

        if(blockchainDTO.getChain().isEmpty()){
            BlockDTO genesisBlock = new
BlockDTO(0,0,null,null,null,null,null,System.currentTimeMillis());
            blockchainRepo.save(genesisBlock);
        }
    }
}
```

6.2.1.2 API testing for the blockchain model

API testing was conducted primarily through Insomnia, which is a popular and widely used API testing tool. Insomnia provides a user-friendly interface that allows developers and testers to create, send, and analyze HTTP requests to test the functionality of APIs. Several APIs were written to assess the functionality of the blockchain model developed by following the aforementioned procedure. The following describes the APIs written to test the blockchain model.

BlockchainController

1. **GET**: /api/blockchain/full
    - **Functionality**: Retrieves the entire blockchain.
    - **Java Method**: getFullBlockchain
    - **HTTP Method**: GET
    - **Return Type**: ArrayList<BlockDTO>

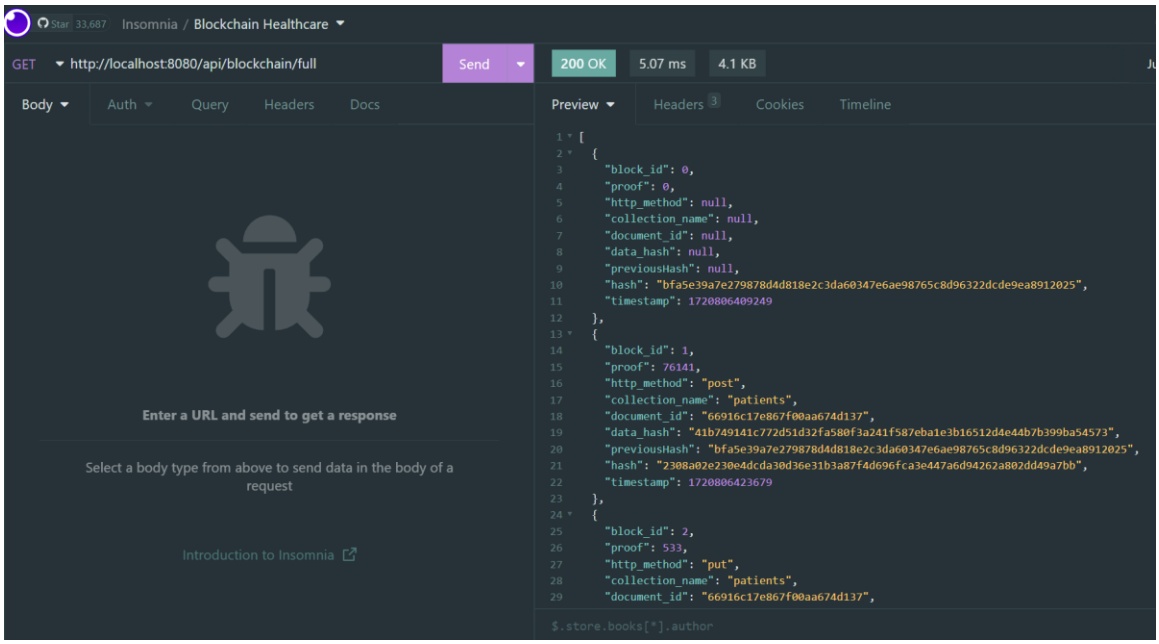


Figure 21: API testing using Insomnia for the getFullblockchain API

2. **GET**: /api/blockchain/validate
    - **Functionality**: Validates the entire blockchain.
    - **Java Method**: validateBlockchain
    - **HTTP Method**: GET
    - **Return Type**: String

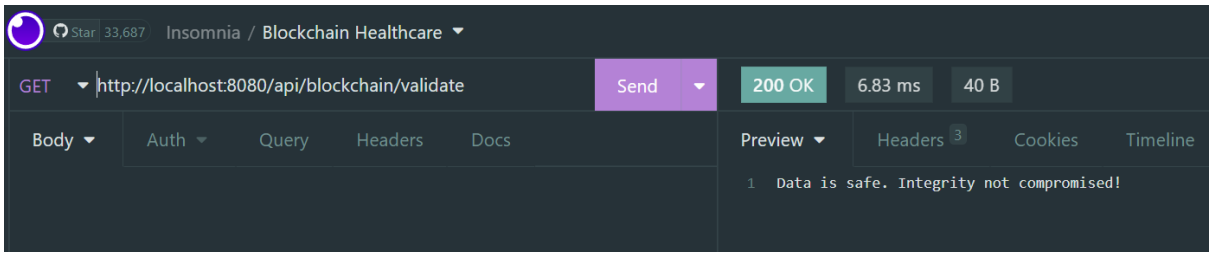


Figure 22: API testing using Insomnia for the ValidateBlockchain API

Lets consider a scenario, a malicious author tampers with the data directly in the database as shown below.

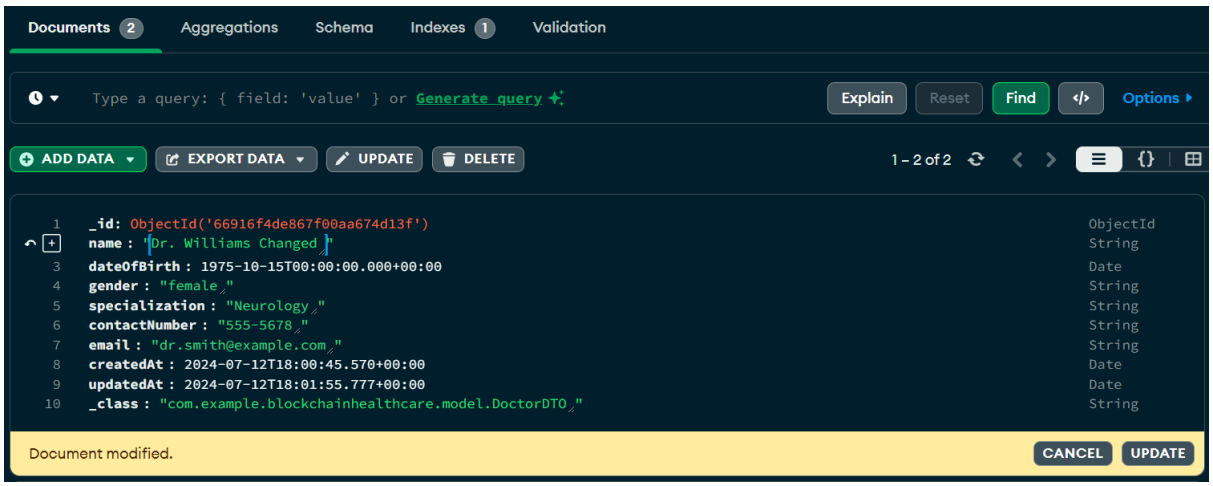


Figure 23: Malicious Author Tampering the Data

As shown in the above diagram, the name of Dr. Williams has been changed. In such a scenario, the expected output from the API will be as follows.

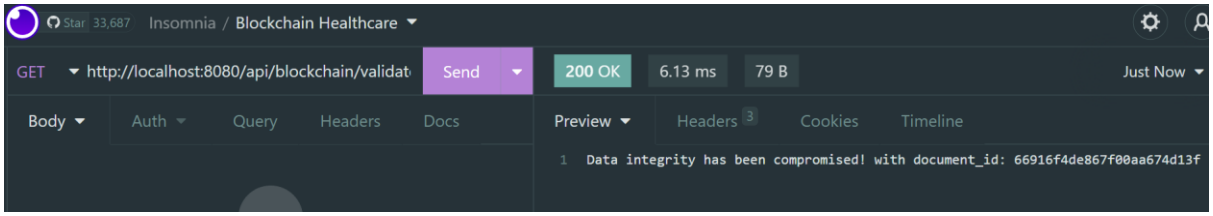


Figure 24: API testing using Insomnia for the ValidateBlockchain API - After Data Been Tampered

DoctorController

1. **GET**: /api/blockchain/doctors
    - **Functionality**: Retrieves all doctors.
    - **Java Method**: getAllDoctors

- **HTTP Method**: GET
- **Return Type**: ResponseEntity<?>

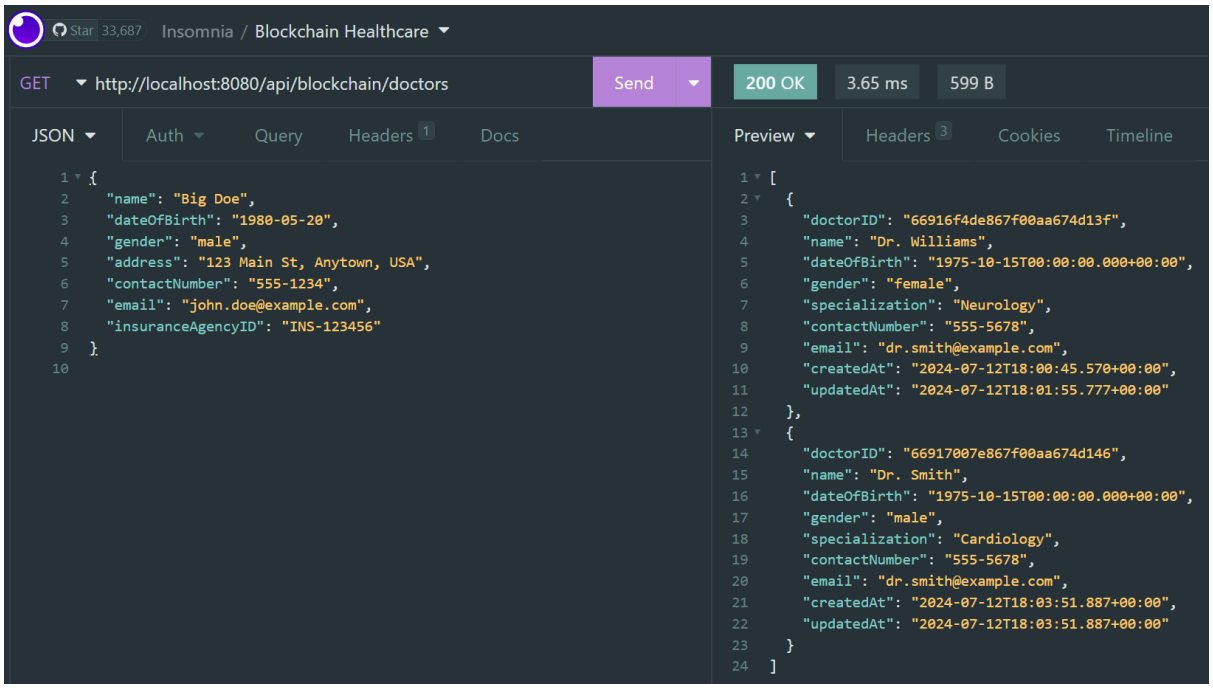


Figure 25: API testing using Insomnia for the getAllDoctors API

2. **GET**: /api/blockchain/doctor/{id}
   - **Functionality**: Retrieves a specific doctor by ID.
   - **Java Method**: getADoctor
   - **HTTP Method**: GET
   - **Return Type**: ResponseEntity<?>

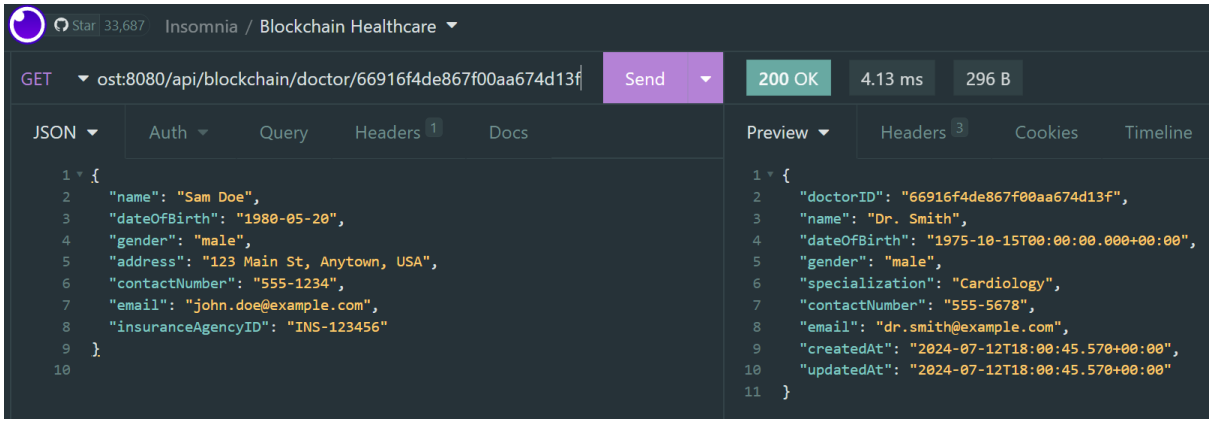


Figure 26: API testing using Insomnia for the get a Doctor API

3. **POST**: /api/blockchain/doctor
   - **Functionality**: Creates a new doctor.
   - **Java Method**: createDoctor
   - **HTTP Method**: POST
   - **Return Type**: ResponseEntity<?>

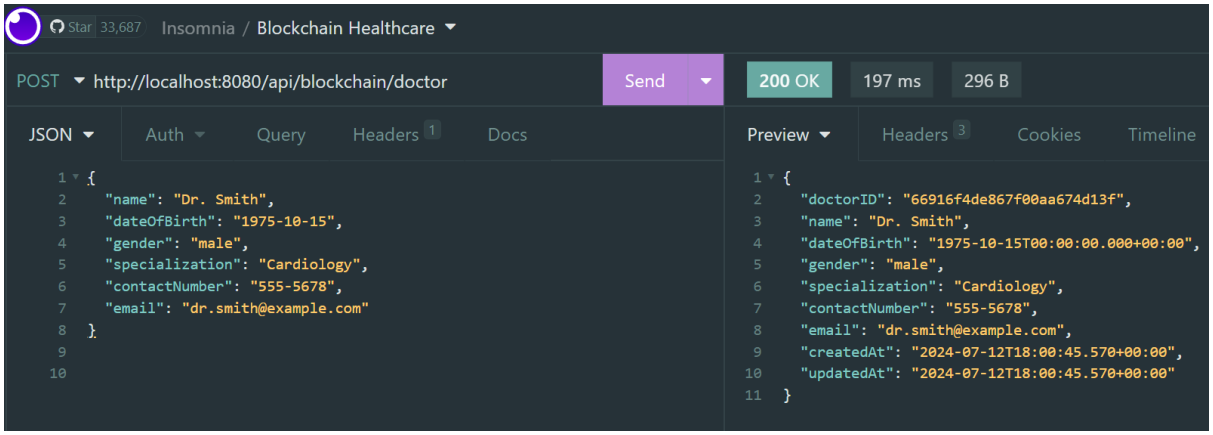


Figure 27: API testing using Insomnia for the create a Doctor API

4. **PUT**: /api/blockchain/doctor/{id}
    - **Functionality**: Updates an existing doctor by ID.
    - **Java Method**: updateADoctor
    - **HTTP Method**: PUT
    - **Return Type**: ResponseEntity<?>

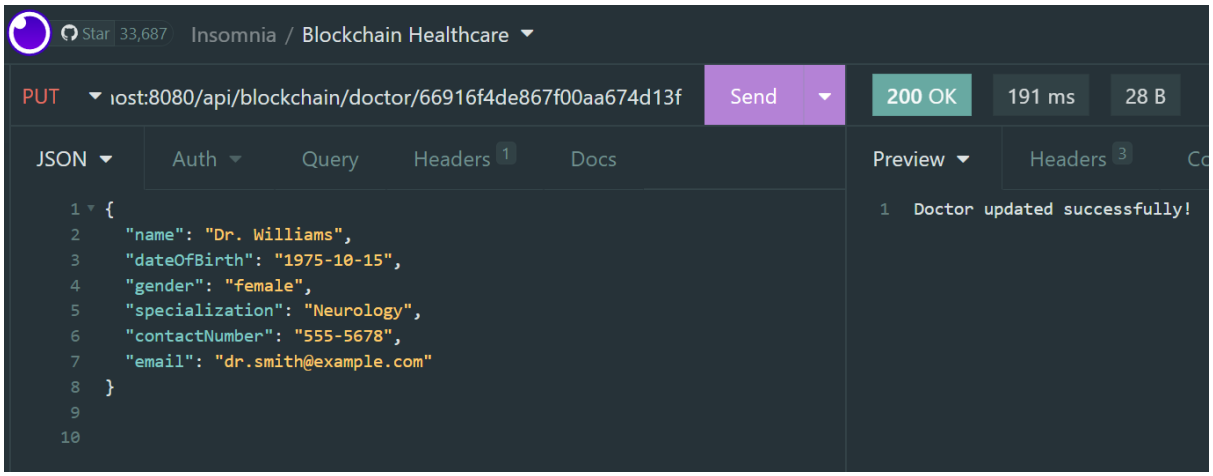


Figure 28: API testing using Insomnia for the update a Doctor API

5. **DELETE**: /api/blockchain/doctor/{id}
    - **Functionality**: Deletes a doctor by ID.
    - **Java Method**: deleteADoctor
    - **HTTP Method**: DELETE
    - **Return Type**: ResponseEntity<?>

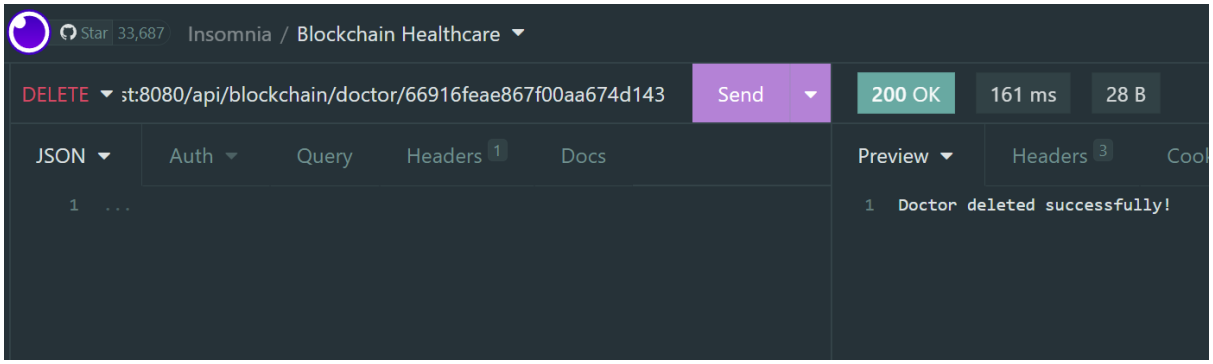


Figure 29: API testing using Insomnia for the delete a Doctor API

PatientController

1. **GET**: /api/blockchain/patients
    - **Functionality**: Retrieves all patients.
    - **Java Method**: getAllPatients
    - **HTTP Method**: GET
    - **Return Type**: ResponseEntity<?>

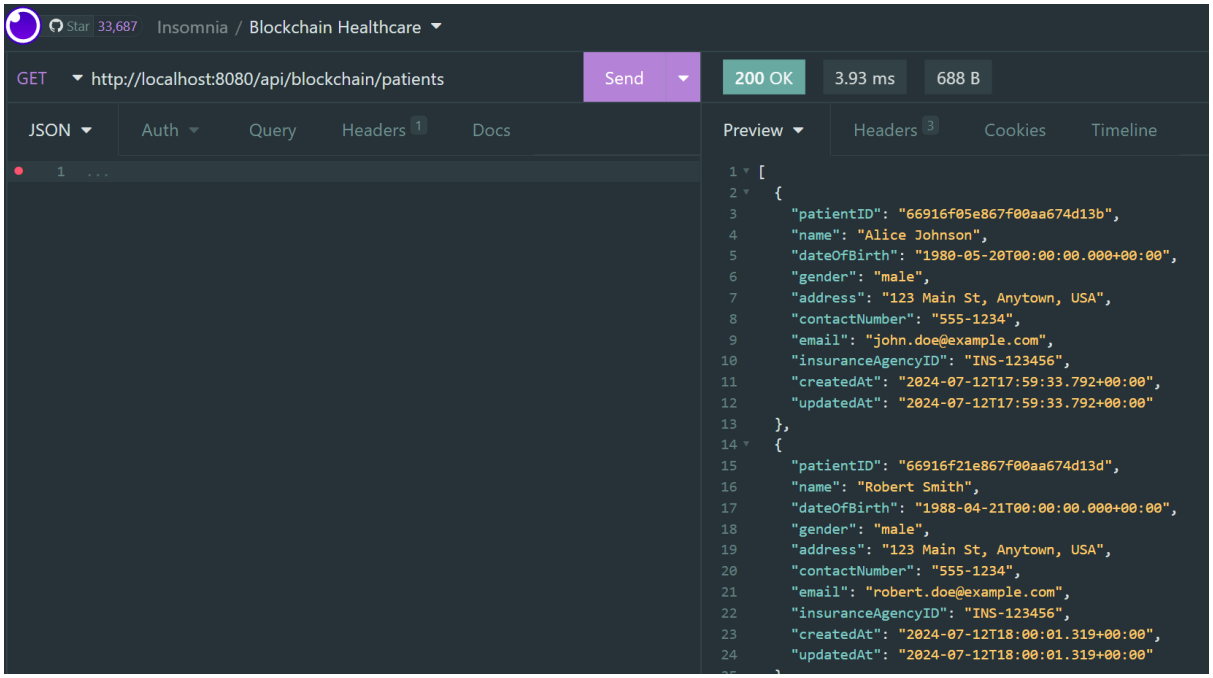


Figure 30: API testing using Insomnia for the getAllPatients API

2. **GET**: /api/blockchain/patient/{id}
    - **Functionality**: Retrieves a specific patient by ID.
    - **Java Method**: getAPatient
    - **HTTP Method**: GET
    - **Return Type**: ResponseEntity<?>

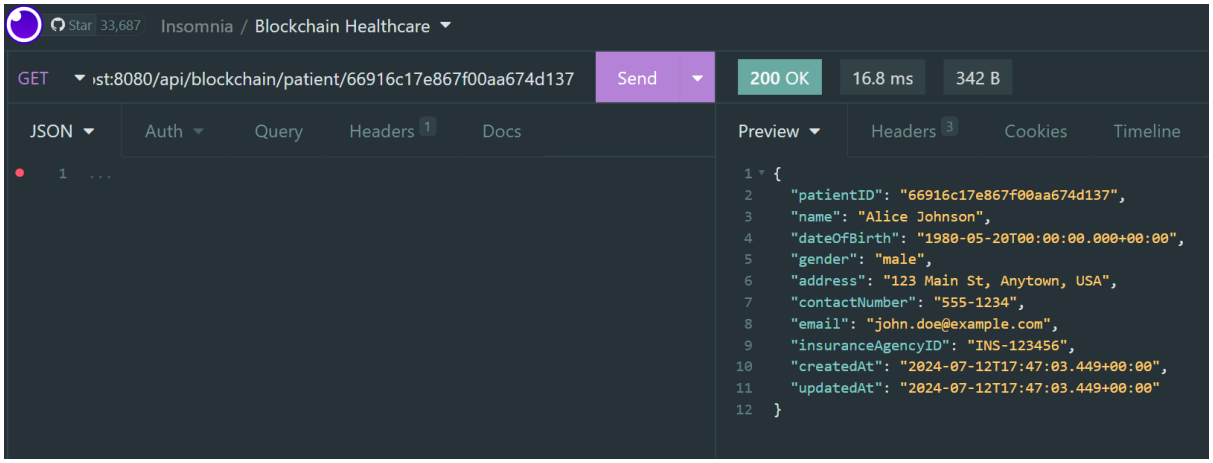


Figure 31: API testing using Insomnia for the get a Patient API

3. **POST**: /api/blockchain/patient
    - **Functionality**: Creates a new patient.
    - **Java Method**: createPatient
    - **HTTP Method**: POST
    - **Return Type**: ResponseEntity<?>

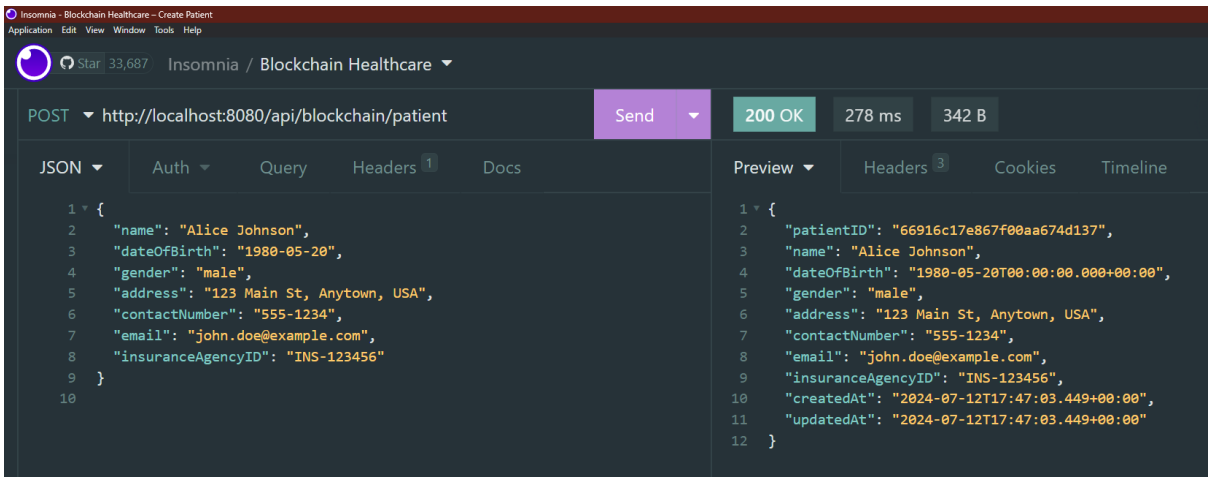


Figure 32: API testing using Insomnia for the create a Patient API

4. **PUT**: /api/blockchain/patient/{id}
    - **Functionality**: Updates an existing patient by ID.
    - **Java Method**: updateAPatient
    - **HTTP Method**: PUT
    - **Return Type**: ResponseEntity<?>

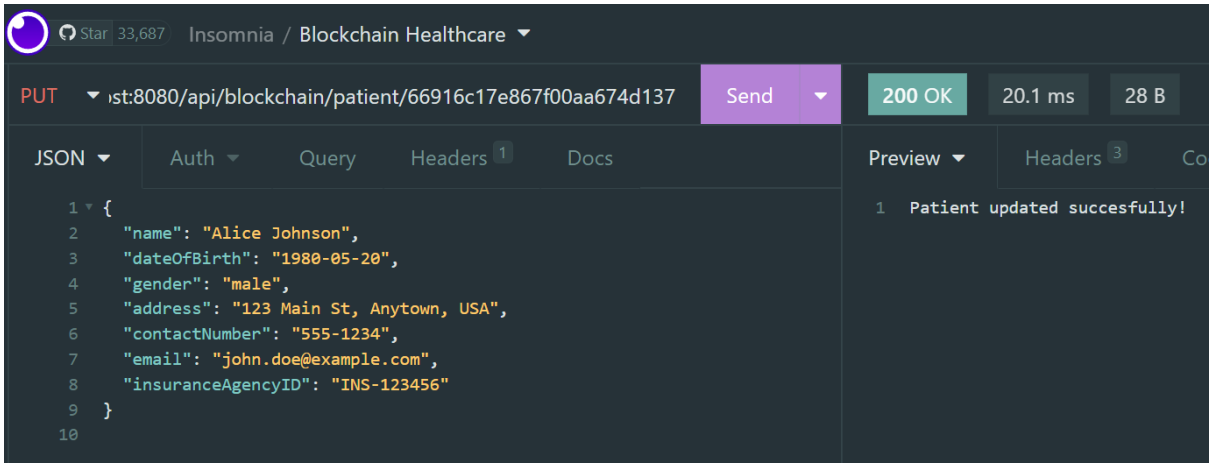


Figure 33: API testing using Insomnia for the update a Patient API

5. **DELETE**: /api/blockchain/patient/{id}
    - **Functionality**: Deletes a patient by ID.
    - **Java Method**: deleteAPatient
    - **HTTP Method**: DELETE
    - **Return Type**: ResponseEntity<?>

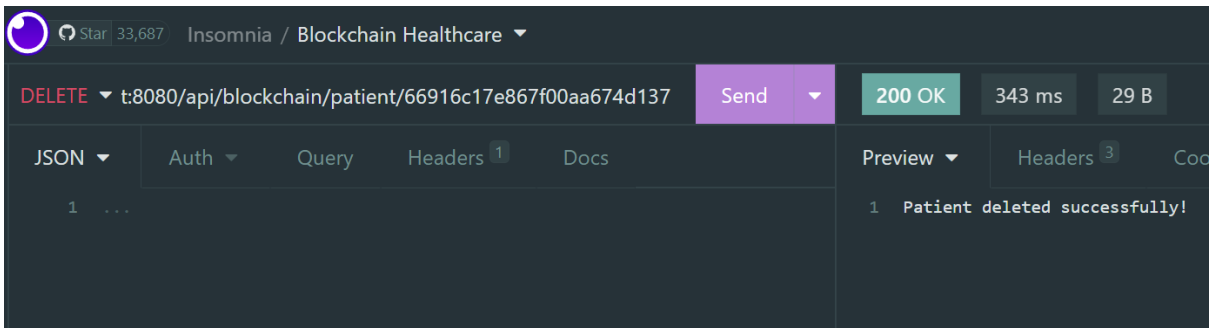


Figure 34: API testing using Insomnia for the delete a Patient APIThe

API testing done for other entities such as Appointments, Lab Reports, Insurance Agents were performed in a similar manner.

6.2.1.3 UI Design & Development

The frontend development was carried out by ReactJS framework. It allows users to directly interact with the API's using a user friendly interface. The Name of the blockchain platform was given as **"Zentura Health"**

Figure 35: Get Started Page

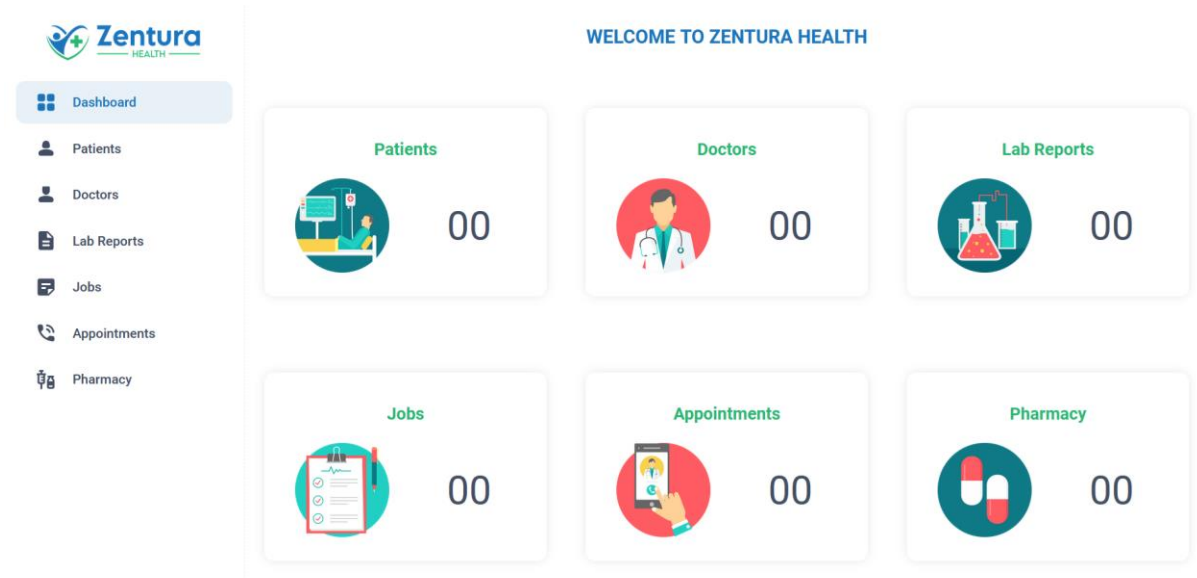


Figure 36: Dashboard Page

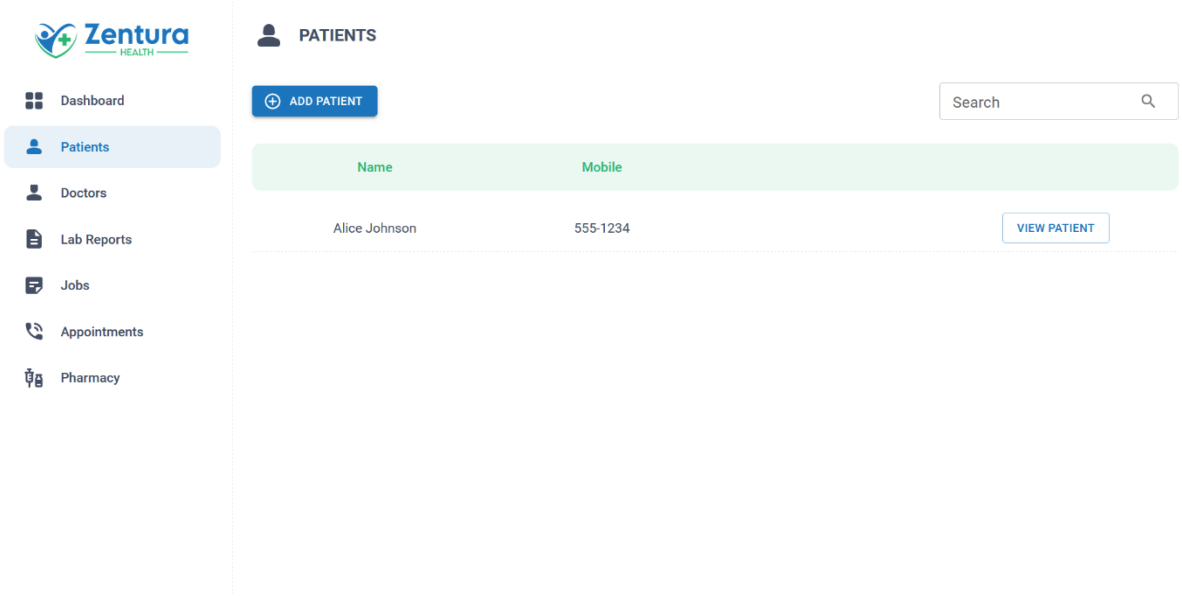


Figure 37: Patients Page

6.2.2 - Module 02: Enhancing Security, Scalability and Reliability of the Blockchain Model

6.2.2.1 Create a custom consensus algorithm fit for the healthcare sector.

As mentioned in the figure xx here is the implementation of the custom consensus algorithm using Java.

1. Modify the BlockchainDTO.java

Add a list of validators and a method to add validators.

```
import java.util.List;

@Getter
@Setter
@Component
public class BlockchainDTO {
    private ArrayList<BlockDTO> chain;
    private List<String> validators;

    // ..

    public void addValidator(String validator) {
        validators.add(validator);
    }
}
```

2. Update BlockchainController.java

Add an API for the validators.

```
@RestController
@RequestMapping("/api/blockchain")
public class BlockchainController {

    // ..

    @PostMapping("/addValidator")
    public void addValidator(@RequestParam String validator) {
        blockchain.addValidator(validator);
    }
}
```

3. Modify mineNewBlock method in BlockchainController.java

Before adding a new block, ensure that the node is a validator.

```
@PostMapping("/mine")
public BlockDTO mineNewBlock(@RequestBody PatientRecordDTO patientRecords,
@RequestParam String validator) {
    if (blockchain.getValidators().contains(validator)) {
        int previousProof = blockchain.getPreviousBlock().getBlock_id();
        int proof = blockchain.proofOfWork(previousProof);
        String previousHash = blockchain.getPreviousBlock().getHash();
        return blockchain.createBlock(proof, previousHash, patientRecords);
    } else {
        throw new IllegalArgumentException("Node is not a validator");
    }
}
```

4. Periodically voting and clear malicious nodes

After a specific period hold a voting assesment to clear the network from bad validators or malicious validators.

```
public void addVote(String validatorId, boolean isTrusted) {
    validatorVotes.put(validatorId,
```

```
validatorVotes.getOrDefault(validatorId, 0) + (isTrusted ? 1 : -1));
}

public void removeMaliciousValidators() {
    validatorVotes.entrySet().removeIf(entry -> entry.getValue() < 0);
}

public void periodicValidatorAssessment() {

    for (String validatorId : validatorVotes.keySet()) {
        boolean isTrusted = assessValidator(validatorId);
addVote(validatorId, isTrusted);
    }
    removeMaliciousValidators();
}
```

5. Proof of Accountability Voting method

Check whether nodes are authorized to add blocks, so after the assesment some nodes may lose their validator role and some may earn their validato role.

```
public String proofOfValidatorIntegrity(int previousProof) {
    if (!isValidatorAuthorized(validatorId)) {
        throw new IllegalStateException("Validator is not authorized to add
blocks");
    }

    periodicValidatorAssessment(); // Integrate periodic assessment in the
PoVI process
    int newProof = 1;
    boolean checkProof = false;

    return Integer.toString(newProof);
}
```

6.2.2.2 API testing of the modified custom consensus algorithm

This demonstration is done by using Postman. To demonstrate this new custom algorithm, we need to add some validators to the system.

1. Add a Validator

Add a validator by passing URL Parameter (hospital)

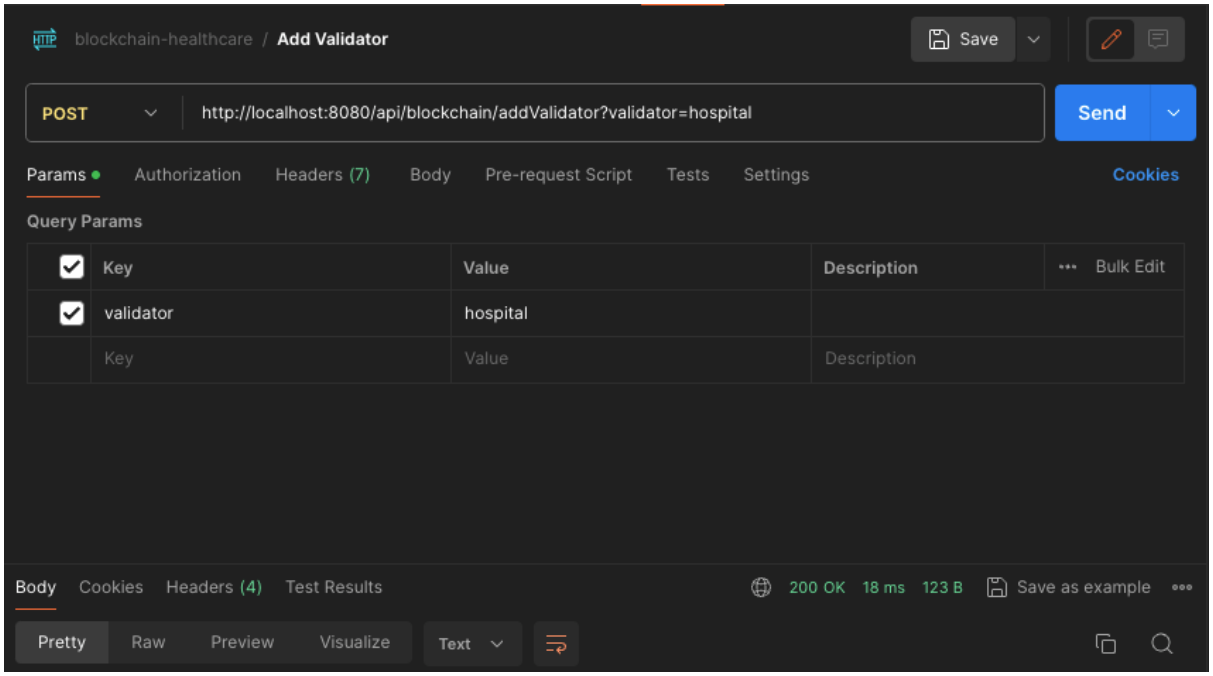


Figure 38: Demonstrate custom consensus algorithm using postman

2. Try to mine a new block (add a new block to the chain) without validating

Get an error (400 Bad Request)

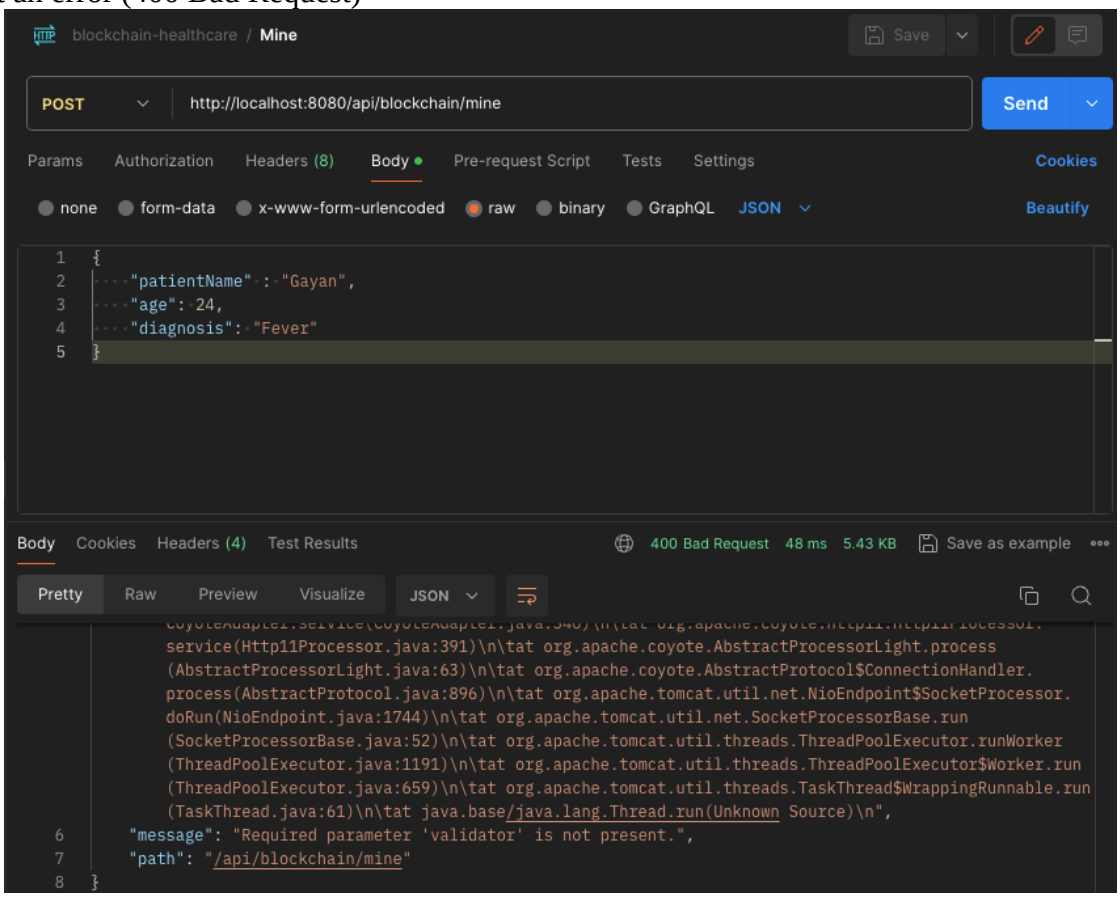


Figure 39: Demonstrate custom consensus algorithm using postman

3. Try to mine a new block with a wrong validator

Try to add a new block and validator is nurse, which is not identified as correct validating node in the system.

Get an error (500 Internal Server Error)

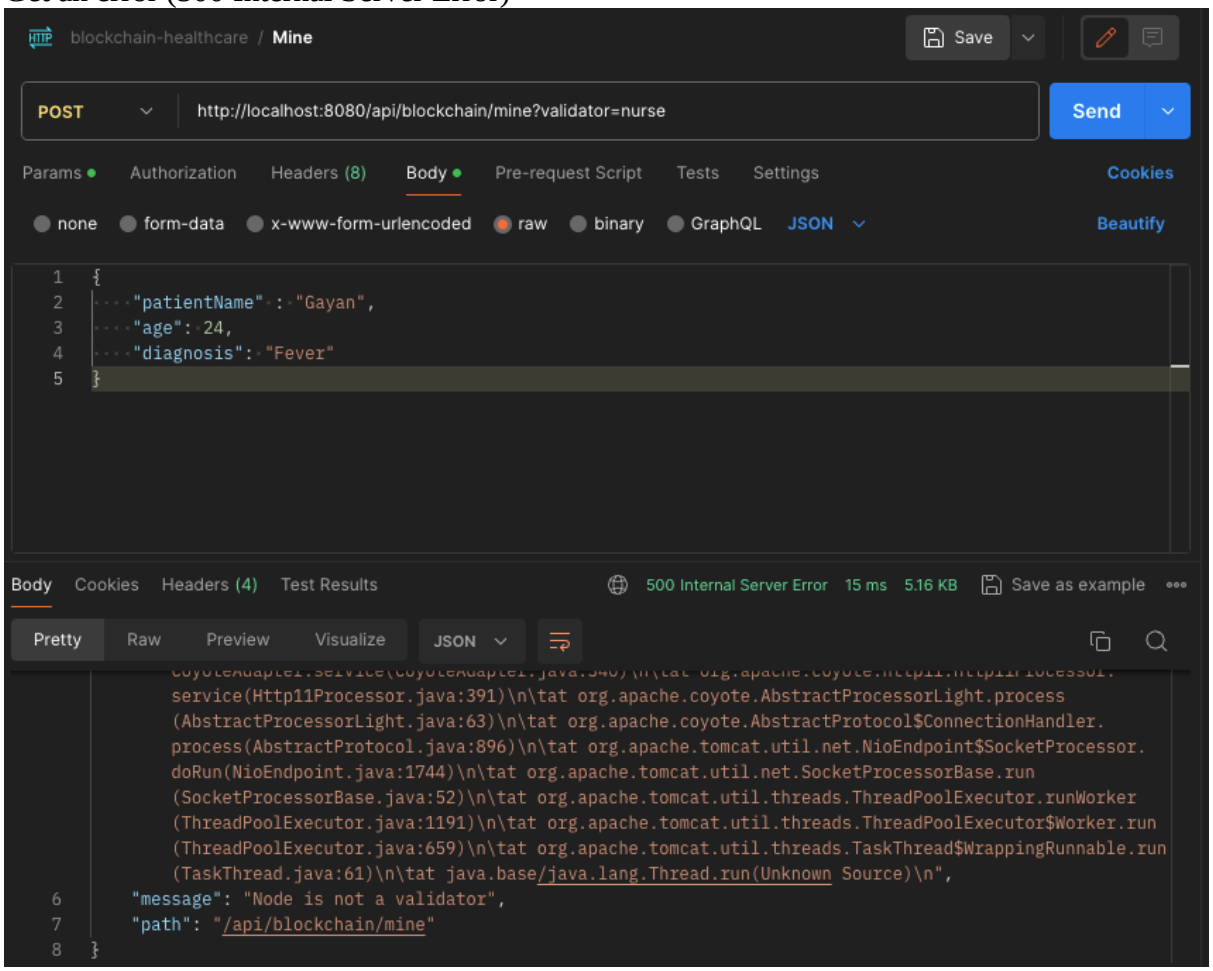


Figure 40: Demonstrate custom consensus algorithm using postman

4. Mine a block with correct validator

Add new block to the chain with a correct validator (validator=hospital)

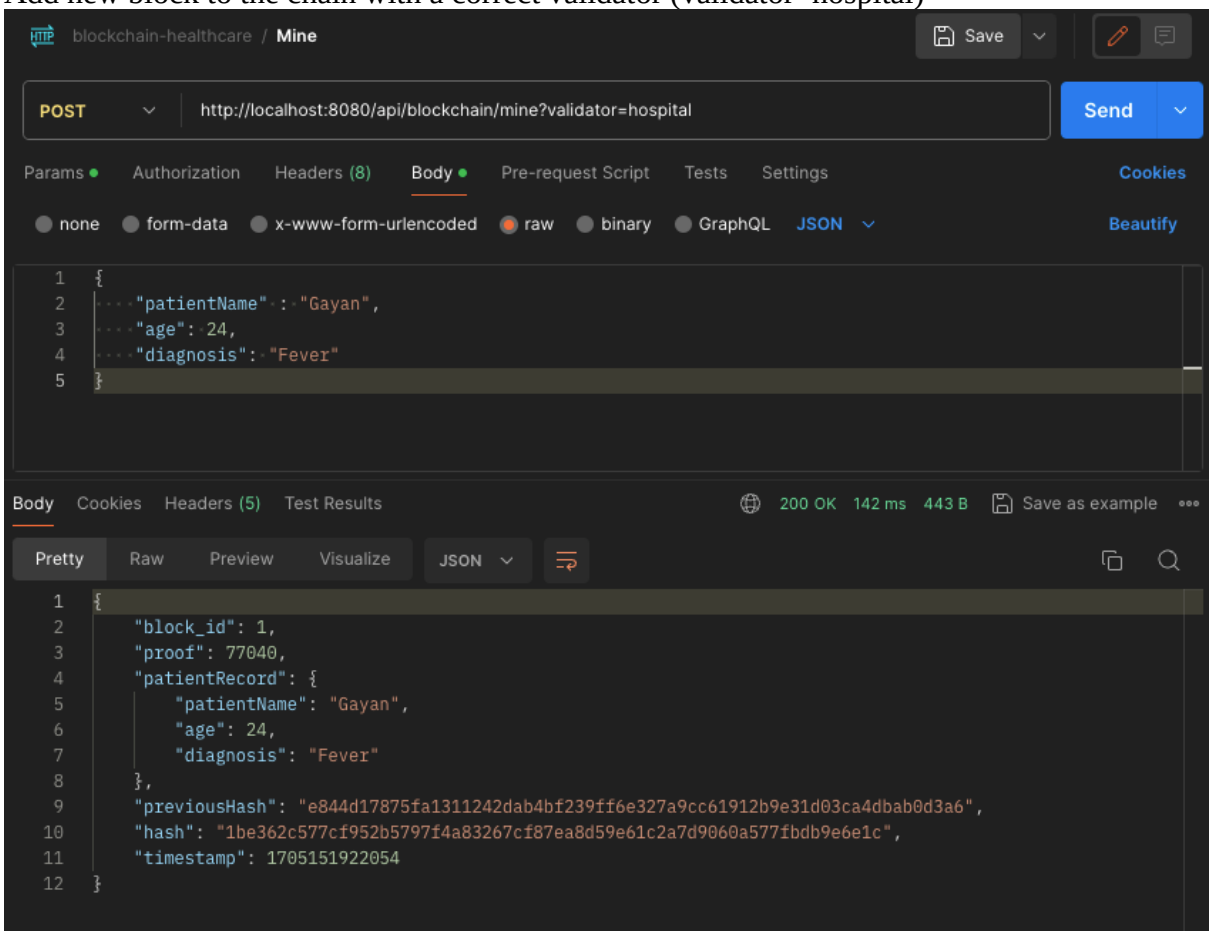


Figure 41: Demonstrate custom consensus algorithm using postman

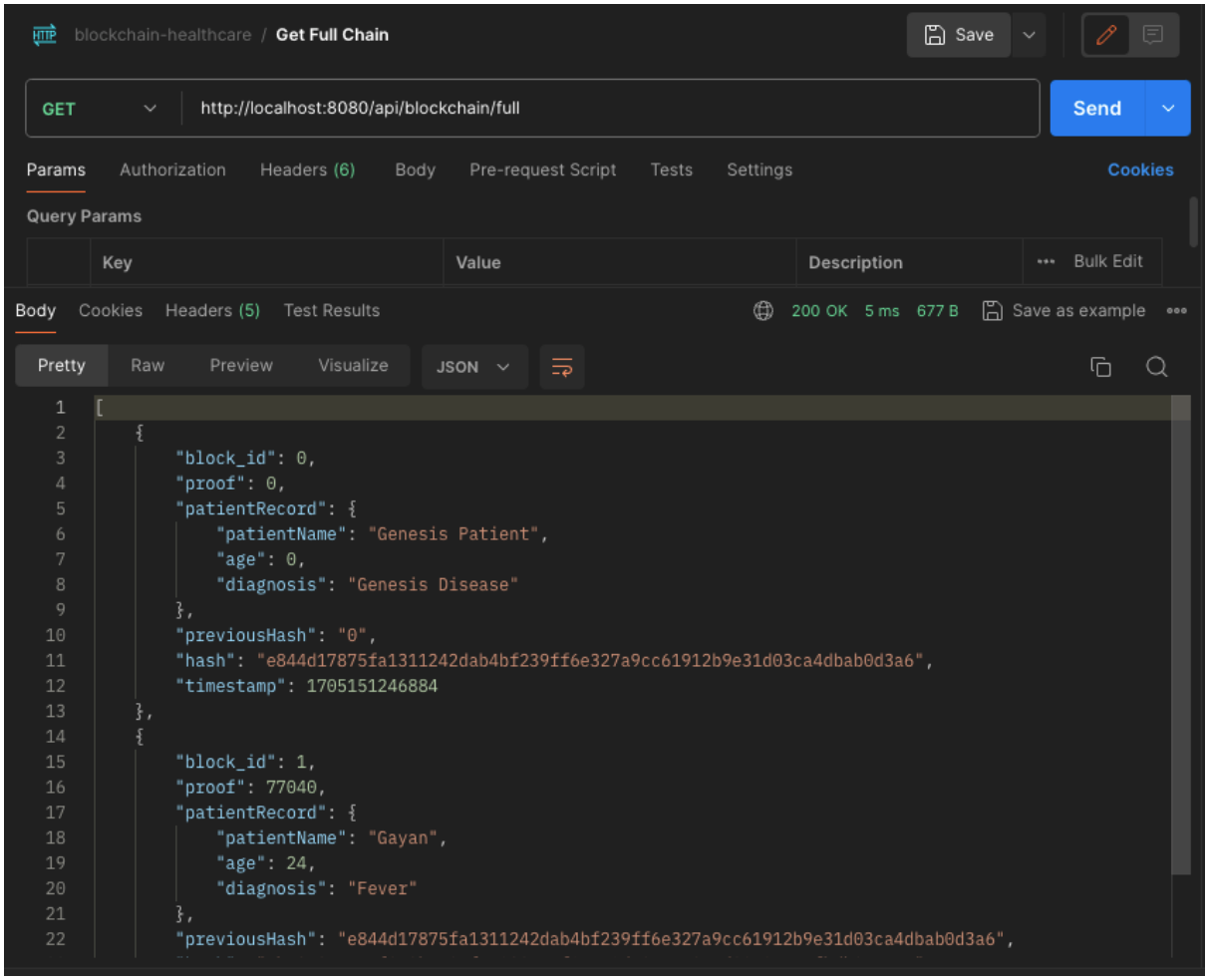
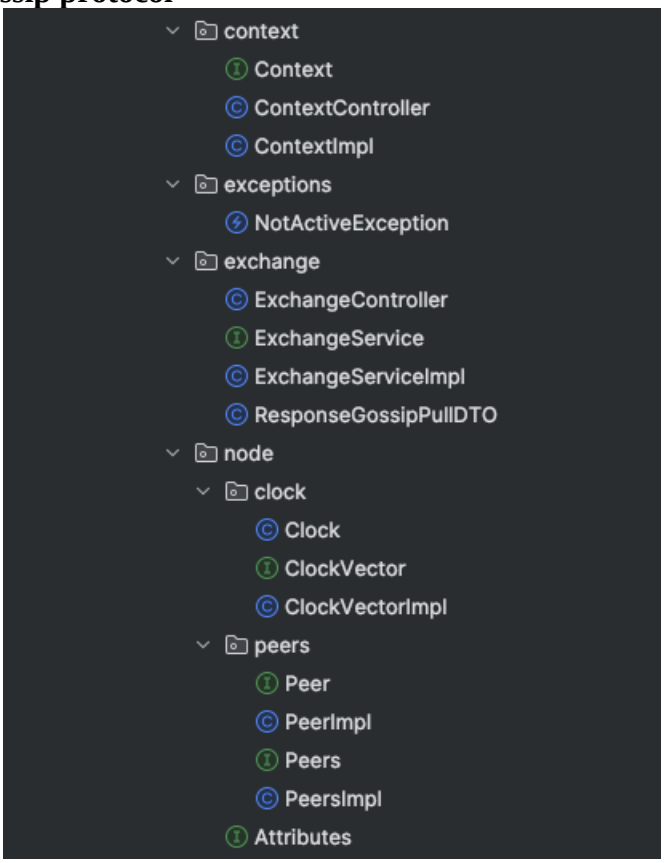


Figure 42: Demonstrate custom consensus algorithm using postman

6.2.2.3 Create a custom blockchain network using gossip protocol fit for the healthcare sector.

1. File tree of the gossip protocol



2. Context Model

```
package com.example.blockchainhealthcare.gossip.context;


import com.example.blockchainhealthcare.gossip.node.clock.Clock;
import com.example.blockchainhealthcare.gossip.node.peers.Peer;

import java.util.List;


public interface Context {

    String getId();

    void setActive(Boolean active);
    Boolean getActive();

    List<Peer> getPeers();
    Peer getPeer(String id);

    Integer getStorageSize();

    List<Clock> getClock();

    void cancelIfNotActive();

}
```

3. Context Controller

```
@RestController
@RequestMapping(value = "/context",produces =
{MediaType.APPLICATION_JSON_VALUE})
@Api(tags="Context")
@RequiredArgsConstructor
class ContextController {

    private final Context context;

    @GetMapping
    @ApiOperation(value = "Get current node meta information")
    public Context getCurrentPeerState()  {
      return context;
    }

    @PostMapping("/stop")
    @ApiOperation(value = "Stop")
    public void stop()  {
        context.setActive(false);
    }

    @PostMapping("/start")
    @ApiOperation(value = "Start")
    public void start()  {
        context.setActive(true);
    }
}
```

4. Context Implementation

If node is failed due to any reason, Node can restart or if you want we can stop the nodes as well

```

@Component
@Slf4j
@RequiredArgsConstructor
class ContextImpl implements Context {

    private final Peers peers;
    private final Attributes attributes;
    private  final ClockVector  clockVector;
    private  final Storage storage;

    @Override
    public String getId() {
        return attributes.getId();
    }

    @Override
    public void setActive(Boolean active) {
        attributes.setActive(active);
    }

    @Override
    public Boolean getActive() {
        return attributes.getActive();
    }

    @Override
    public List<Peer> getPeers() {
```

```
        return peers.getPeers();
    }

    @Override
    public Peer getPeer(String id)  {
        return  peers.get(id);
    }

    @Override
    public List<Clock> getClock() {
        return clockVector.get();
    }

    @Override
    public void cancelIfNotActive() {
        attributes.cancelIfNotActive();
    }

    @Override
    public Integer getStorageSize()  {
        return storage.all().size();
    }


}
```

5. Exchange Controller

```
@RestController
@RequestMapping(value = "/gossip",produces =
{MediaType.APPLICATION_JSON_VALUE})
@Api(tags="Gossip")
@RequiredArgsConstructor
class ExchangeController {

    private final ExchangeService exchangeService;

    @GetMapping
    @ApiOperation(value = "Gossip pull")
    public ResponseGossipPullDTO gossip(@RequestParam String id,
@RequestParam Integer version)
    {
      return exchangeService.gossipPullResponse(id, version);
    }

}
```

6. Exchange Service Implementation

Here, each node send a pull request from each other nodes and always keep in touch with other nodes of the blockchain network, then if any node failure quickly identify the failure node and if block added to a node, other nodes can quickly pull it.

```

@Service
@Slf4j
@RequiredArgsConstructor
class ExchangeServiceImpl implements ExchangeService {

    private final Attributes attributes;
    private final Peers peers;
    private final ClockVector clockVector;
    private final Http http;
```

```
    private final StorageService storageService;



    private CompletableFuture<ResponseGossipPullDTO>
sendPullForOnePeer(String idPeer) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                log.debug("Peer #{} send pull request to  {}",
attributes.getId(), idPeer);
                MultiValueMap<String, String> params = new
LinkedMultiValueMap<>();
                params.add("id", attributes.getId());
                params.add("version",
clockVector.getPeerVersion(idPeer).toString());

                ResponseEntity<ResponseGossipPullDTO> response =
http.callGet(idPeer,
                        ResponseGossipPullDTO.class,
                        params,
                        "gossip");
                return
Optional.ofNullable(response.getBody()).orElse(null);
            } catch (HttpException e) {
                log.error("Peer #{} pull request error for {}. Response
status code {}", attributes.getId(),
                        idPeer, e.getStatusCode());
                return null;
            } catch (ResourceAccessException e) {
                log.error("Peer #{} pull request error for {}. {} {} ",
attributes.getId(), idPeer, e.getClass(),
                        e.getMessage());
                return null;
            } catch (Exception e) {
                log.error(String.format("Peer #%d pull request error for
%d", attributes.getId(), idPeer), e);
                return null;
            }
        });
    }


    @Override
    public void gossipPull() {
        ResponseGossipPullDTO response =
sendPullForOnePeer(peers.getRandom().getId()).join();
        if (response != null) {
            String idPeer = response.getIdPeer();
            log.debug("Peer #{} process request from {}",
attributes.getId(), idPeer);
            if (response.getRecords().size() > 0) {
                log.debug("Peer #{} get data from {} version {} record
count {} ", attributes.getId(),
                        idPeer, response.getVersion(),
response.getRecords().size());
                boolean incVersion = false;
                for (Record record : response.getRecords()) {
                    if (storageService.add(record)) {
                        clockVector.incPeerVersion(idPeer);
                        incVersion = true;
                    }
```

```
                }
                if (incVersion) {
                    clockVector.incCurrVersion();
                }
                while (clockVector.getPeerVersion(idPeer) <
Math.min(response.getVersion(), clockVector.getCurrVersion() - 1))
                    clockVector.incPeerVersion(idPeer);
            }
        }
    }


    @Override
    public ResponseGossipPullDTO gossipPullResponse(String id,
                                                    Integer oldPeerVersion)
{

        log.debug("Peer #{} get pull request from {} version {}",
attributes.getId(), id, oldPeerVersion);
        attributes.cancelIfNotActive();

        Integer currVersion = clockVector.getCurrVersion();
        Integer peerVersion = clockVector.getPeerVersion(id);

        List<Record> records = new ArrayList<>();

        log.debug("Peer #{} check pull request. Version {}, peer version
{}, current version {} ", attributes.getId(),
                oldPeerVersion, peerVersion, currVersion);
        if ((oldPeerVersion < currVersion) && (currVersion > peerVersion +
1 || peerVersion == 0)) {
            log.debug("Peer #{} prepare data to answer {}.  Version {},
peer version {}, current version {} ",
                    attributes.getId(), id, oldPeerVersion, peerVersion,
currVersion);
            List<Record> storage = storageService.all();
            for (int i = storage.size() - 1; i >= 0 &&
storage.get(i).getVersion() > oldPeerVersion; i--) {
                records.add(storage.get(i));
            }
        }
        return new ResponseGossipPullDTO(attributes.getId(), currVersion,
records);

    }


}
```

7. Gossip Timer

In this class, Gossip Timer helps to check and decide whether other nodes are active or not, wait and check certain time and if no response from the other node, with help of gossip timer we can decide that node is currently not active.

```
package com.example.blockchainhealthcare.gossip.timer;


import com.example.blockchainhealthcare.gossip.exchange.ExchangeService;
import com.example.blockchainhealthcare.gossip.node.Attributes;
import lombok.Getter;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;
import java.util.Timer;
import java.util.TimerTask;
import java.util.concurrent.atomic.AtomicInteger;


@Slf4j
@RequiredArgsConstructor
@Component
public class GossipTimer {

    private final Attributes attributes;
    private final Timer timer = new Timer();
    private final ExchangeService exchangeService;


    @Getter
    private final AtomicInteger counter = new AtomicInteger(0);



    @Value("${gossip.timeout}" )
    Integer timeout;


    @PostConstruct
    private void start() {

        timer.schedule(new TimerTask() {
            public void run() {

                if (attributes.getActive()) {

                    counter.incrementAndGet();
                    log.debug("Peer #{} Time to next gossip: {} sec",
attributes.getId(),timeout - counter.get());
                    if (counter.get() >=  timeout) {
                        counter.set(0);
                        exchangeService.gossipPull();
                    }
                }
                else
                    counter.set(0);

            }
```

```
        }, 0, 1000);
    }


}
```

6.2.3 - Module 03: Streamlining Patient Data Consent Management Processes

6.2.3.1 Role Based Access Control System

Building a Role-Based Access Control (RBAC) system with JSON Web Tokens (JWT) in Spring Boot involves integrating security mechanisms to manage user roles and permissions effectively. JWT serves as a secure means of authentication and authorization within the application, ensuring that only authorized users can access specific resources based on their assigned roles. This implementation involves configuring Spring Security to validate JWT tokens and enforce access control policies based on user roles defined in the system.

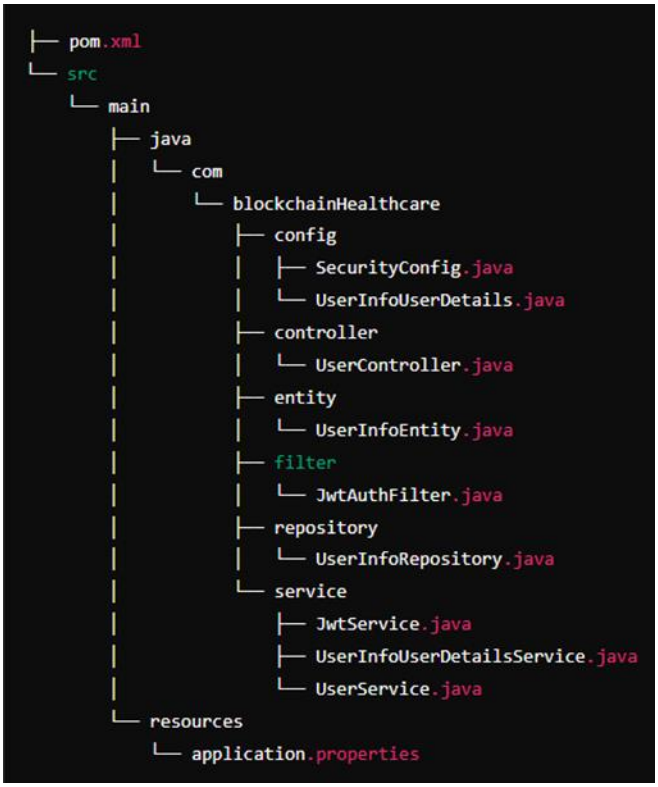


Figure: Directory structure of the RBAC System

The figure shows the file structure of the Spring Boot application for healthcare blockchain implementation ensures a clear separation of concerns and efficient organization of components. Key components such as **SecurityConfig.java** and **UserInfoUserDetails.java** in the config directory handle Spring Security configurations and provide implementations for user authentication and authorization through RBAC. Supporting components such as controllers, Services, Repositories are structured to manage various functionalities like user management, JWT token handling, and data access. This structured approach ensures that access to sensitive medical data is secured according to defined roles and permissions,

implemented using Spring Security's RBAC mechanisms, and organized efficiently for scalability and maintainability.

1. **Spring Security Configuration**

The `SecurityConfig.java` file is a central configuration class in a Spring Boot application responsible for defining security configurations using Spring Security

- **Security Filter Chain**:
  - securityFilterChain(HttpSecurity http):
    - Configures the security filter chain using HttpSecurity builder.
  - .csrf(csrf -> csrf.disable()):
    - Disables CSRF (Cross-Site Request Forgery) protection as it's not needed for stateless authentication with JWT.
  - .authorizeHttpRequests(req -> req...):
    - Configures URL-based authorization rules, specifying which endpoints are accessible to different types of users based on user role.

```
.authorizeHttpRequests(req -> req
        .requestMatchers("/api/blockchain/new-user").permitAll()
        .requestMatchers("/api/blockchain/authenticate").permitAll()
        .requestMatchers("/api/blockchain/full").permitAll()
        .requestMatchers("/api/blockchain/valid").authenticated()
        .requestMatchers("/api/blockchain/mine").authenticated()
        .requestMatchers("/api/blockchain/patient").authenticated()
)
```

Authority Restrictions

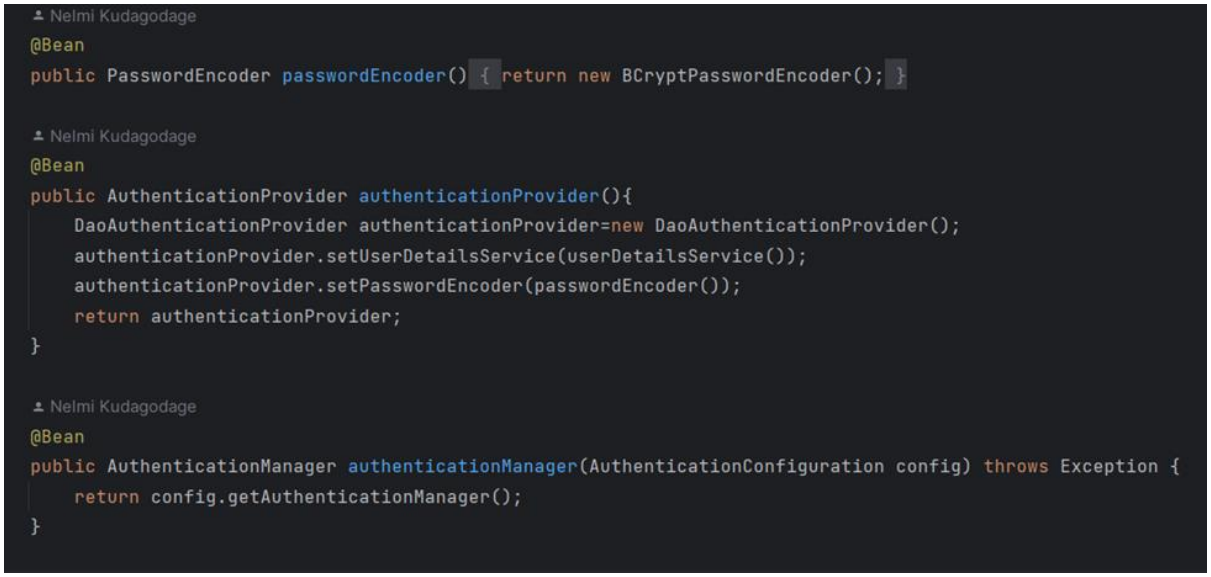
  - .sessionManagement(session -> session...):
    - Configures session management to use stateless sessions, ensuring no session data is stored on the server.
  - .authenticationProvider(authenticationProvider()):
    - Sets the custom DaoAuthenticationProvider to handle authentication.
  - .addFilterBefore(authFilter, UsernamePasswordAuthenticationFilter.class):
    - Registers the JwtAuthFilter before the UsernamePasswordAuthenticationFilter for processing JWT tokens.

```java
@EnableMethodSecurity
public class SecurityConfig {

    @Autowired
    private JwtAuthFilter authFilter;
    Nelmi Kudagodage *
    @Bean
    //authentication
    public UserDetailsService userDetailsService() {
        return new UserInfoUserDetailsService();
    }
    Nelmi Kudagodage *
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http
                .csrf(csrf -> csrf.disable())
                .authorizeHttpRequests(req -> req
                        .requestMatchers("/api/blockchain/new-user").permitAll()
                        .requestMatchers("/api/blockchain/authenticate").permitAll()
                        .requestMatchers("/api/blockchain/full").permitAll()
                        .requestMatchers("/api/blockchain/valid").authenticated()
                        .requestMatchers("/api/blockchain/mine").authenticated()
                        .requestMatchers("/api/blockchain/patient").authenticated()
                )
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                )
                .authenticationProvider(authenticationProvider())
                .addFilterBefore(authFilter, UsernamePasswordAuthenticationFilter.class)
                .build();
    }
```

JWT Auth Filter

- **Authentication Provider**:
    - authenticationProvider():
        - Configures a DaoAuthenticationProvider that uses the UserDetailsService and PasswordEncoder beans for authentication.

- **Authentication Manager**:
    - authenticationManager(AuthenticationConfiguration config):
        - Defines an AuthenticationManager bean to manage authentication processes.



Authentication Provider and Authentication Manager

2. JWT Authentication Filter

The JwtAuthFilter class is a custom filter implemented in a Spring Boot application to handle JWT based authentication for securing HTTP requests.

- **Extending OncePerRequestFilter**:
  - extends OncePerRequestFilter:
    - Extends Spring's OncePerRequestFilter class, ensuring that the doFilterInternal method is invoked only once per request.
- **doFilterInternal Method**:
  - doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain):
    - Overrides the method to implement custom logic for JWT authentication and authorization.
- **JWT Authentication Logic**:
  - Retrieves the JWT token from the HTTP request header ("Authorization").
  - Extracts the username from the JWT token using jwtService.extractUsername(token).
  - Checks if a valid username is extracted and if there's no existing authentication context (SecurityContextHolder.getContext().getAuthentication()).
  - Loads user details (UserDetails) from the database using userDetailsService.loadUserByUsername(username).
  - Validates the JWT token against the loaded user details (userDetails) using jwtService.validateToken(token, userDetails).
  - If the token is valid, creates an UsernamePasswordAuthenticationToken instance with the user details and sets it in the SecurityContextHolder.
  - authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request)) sets additional details about the authentication request (like IP address, session ID) to authToken.

- **Filter Chain Handling**:
    - filterChain.doFilter(request, response):
        - Continues the filter chain, allowing the request to proceed to the next filter or handler in the Spring MVC flow after JWT authentication.

```
@Component
public class JwtAuthFilter extends OncePerRequestFilter {

    @Autowired
    private JwtService jwtService;

    @Autowired
    private UserInfoUserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
response, FilterChain filterChain) throws ServletException, IOException {
        String authHeader = request.getHeader("Authorization");
        String token = null;
        String username = null;
        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            token = authHeader.substring(7);
            username = jwtService.extractUsername(token);
        }

        if (username != null && SecurityContextHolder.getContext().getAuthentication()
== null) {
            UserDetails userDetails = userDetailsService.loadUserByUsername(username);
            if (jwtService.validateToken(token, userDetails)) {
                UsernamePasswordAuthenticationToken authToken = new
UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());
                authToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

3. JWT Service

The JwtService class is a component in a Spring Boot application responsible for generating, extracting, and validating JWT tokens.

- Token Extraction Methods:

    - extractUsername(String token):
        - Extracts the username (subject) from the JWT token by calling the extractClaim method with Claims::getSubject.
    - extractExpiration(String token):
        - Extracts the expiration date of the JWT token by calling the extractClaim method with Claims::getExpiration.

- Claims Extraction Methods:
    - extractClaim(String token, claimsResolver):
        - A generic method to extract any claim from the JWT token using a provided claims resolver function. It retrieves all claims and applies the resolver function to get the specific claim.
    - private Claims extractAllClaims(String token):
        - Parses the JWT token and extracts all claims. It uses the secret key to validate the token's signature and returns the claims.
- Token Validation Methods:
    - private Boolean isTokenExpired(String token):
        - Checks if the JWT token has expired by comparing its expiration date with the current date.
    - public Boolean validateToken(String token, UserDetails userDetails):
        - Validates the JWT token by checking if the username in the token matches the username in the UserDetails object and ensuring the token is not expired.
- Token Generation Methods:
    - public String generateToken(String userName):
        - Generates a JWT token for a given username. It creates an empty claims map and calls the createToken method.
    - private String createToken(Map<String, Object> claims, String userName):
        - Builds the JWT token with the provided claims, subject (username), issue date, and expiration date. It signs the token using the secret key and the HS256 algorithm.
- Signing Key Method:
    - private Key getSignKey():
        - Decodes the secret key from its Base64 encoded form and returns a Key object for signing the JWT tokens.

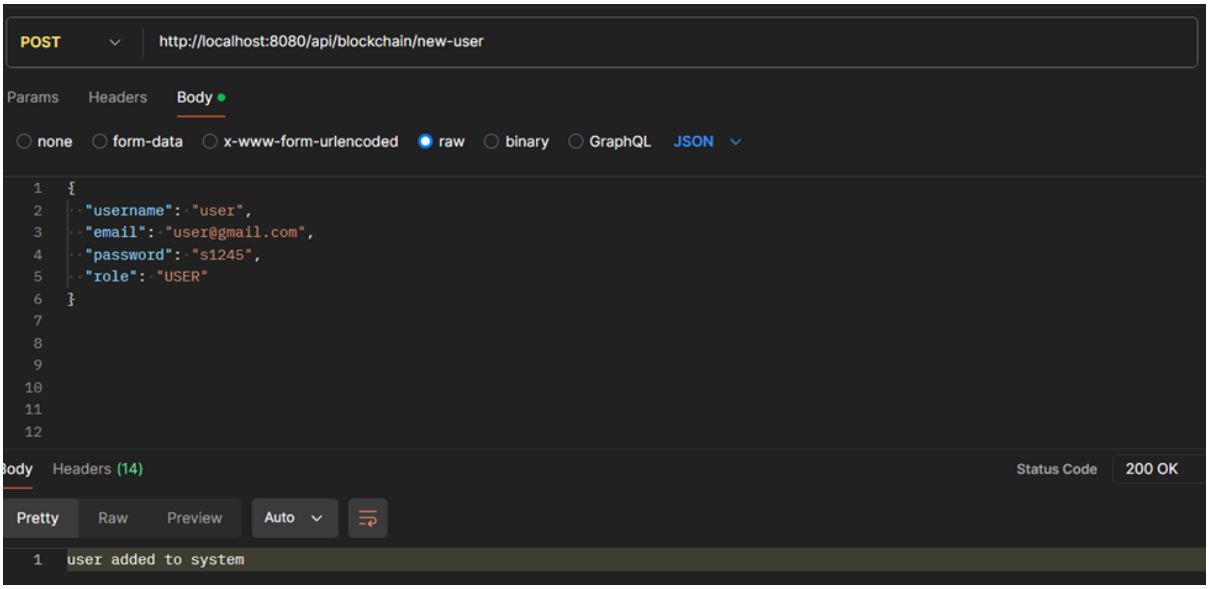
```
@Component
public class JwtService {



    public static final String SECRET =
"5367566B59703373367639792F423F4528482B4D6251655468576D5A71347437";



    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

    public Date extractExpiration(String token) {
        return extractClaim(token, Claims::getExpiration);
    }

    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }
```

```
    private Claims extractAllClaims(String token) {
        return Jwts
                .parserBuilder()
                .setSigningKey(getSignKey())
                .build()
                .parseClaimsJws(token)
                .getBody();
    }

    private Boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    public Boolean validateToken(String token, UserDetails userDetails) {
        final String username = extractUsername(token);
        return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
    }



    public String generateToken(String userName){
        Map<String,Object> claims=new HashMap<>();
        return createToken(claims,userName);
    }

    private String createToken(Map<String, Object> claims, String userName) {
        return Jwts.builder()
                .setClaims(claims)
                .setSubject(userName)
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis()+1000*60*30))
                .signWith(getSignKey(), SignatureAlgorithm.HS256).compact();
    }

    private Key getSignKey() {
        byte[] keyBytes= Decoders.BASE64.decode(SECRET);
        return Keys.hmacShaKeyFor(keyBytes);
    }
}
```

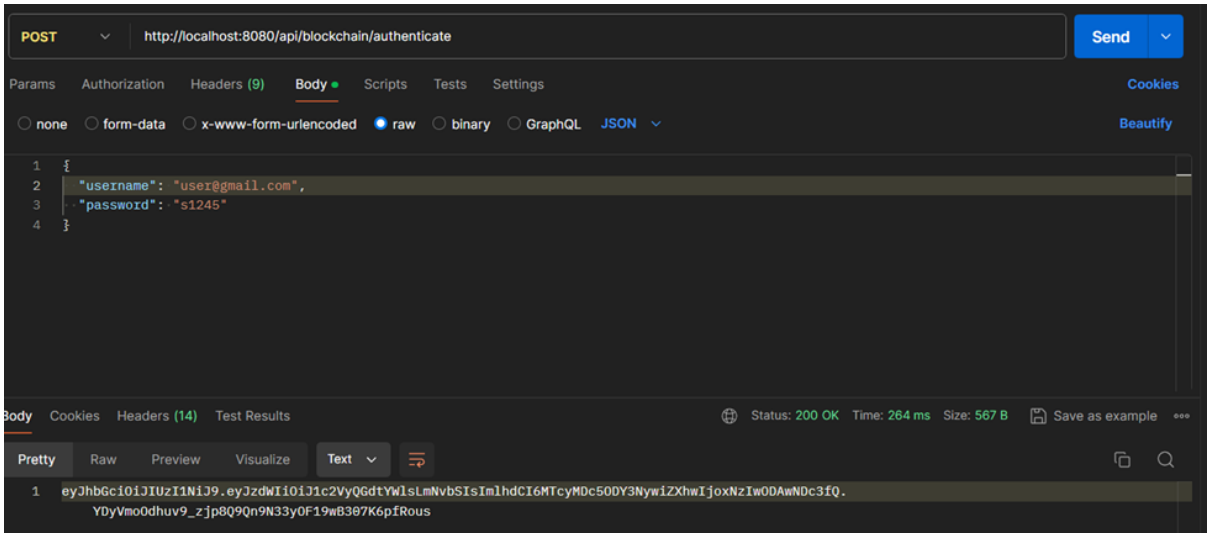
6.2.3.2 API testing of the modified RBAC

1. Add a new user to blockchain

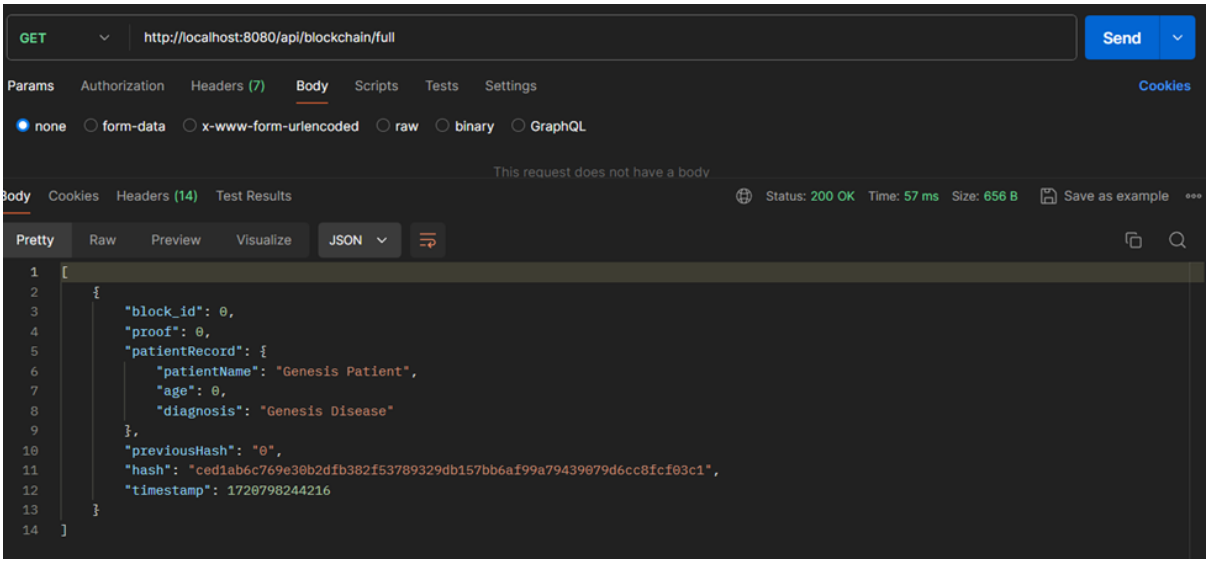
   Add a new user to the system by using "http://localhost:8080/api/blockchain/new-user" API and will assign role as 'USER'



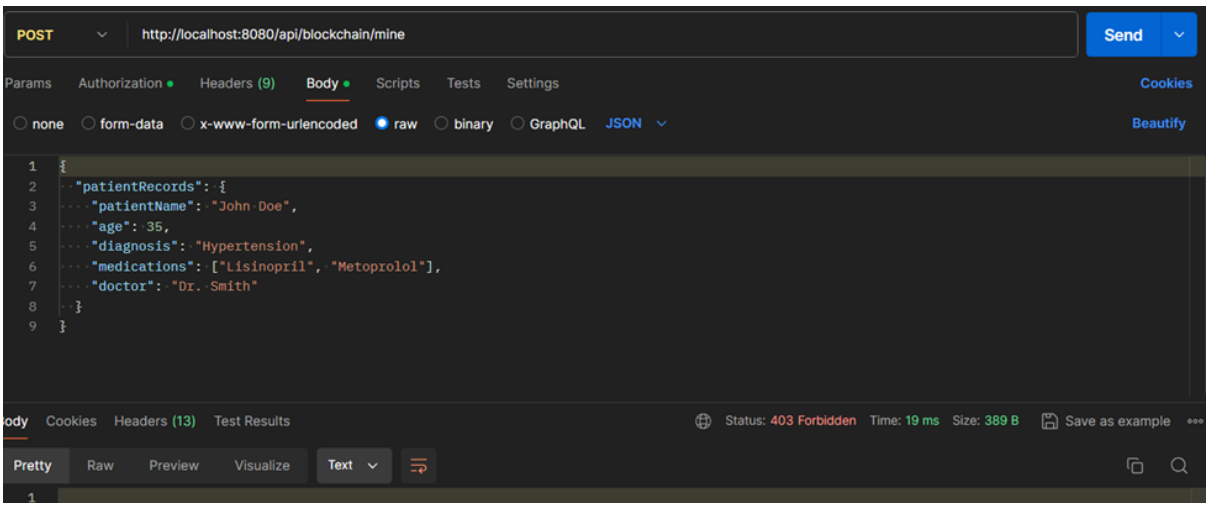
2. Role “User” have access to /api/blockchain/authenticate API endpoint. Let’s login to system using /authticate endpoint and try to retrieve JWT token for access other endpoints.



3. By using generated JWT token let’s try to access different endpoints in blockchain.

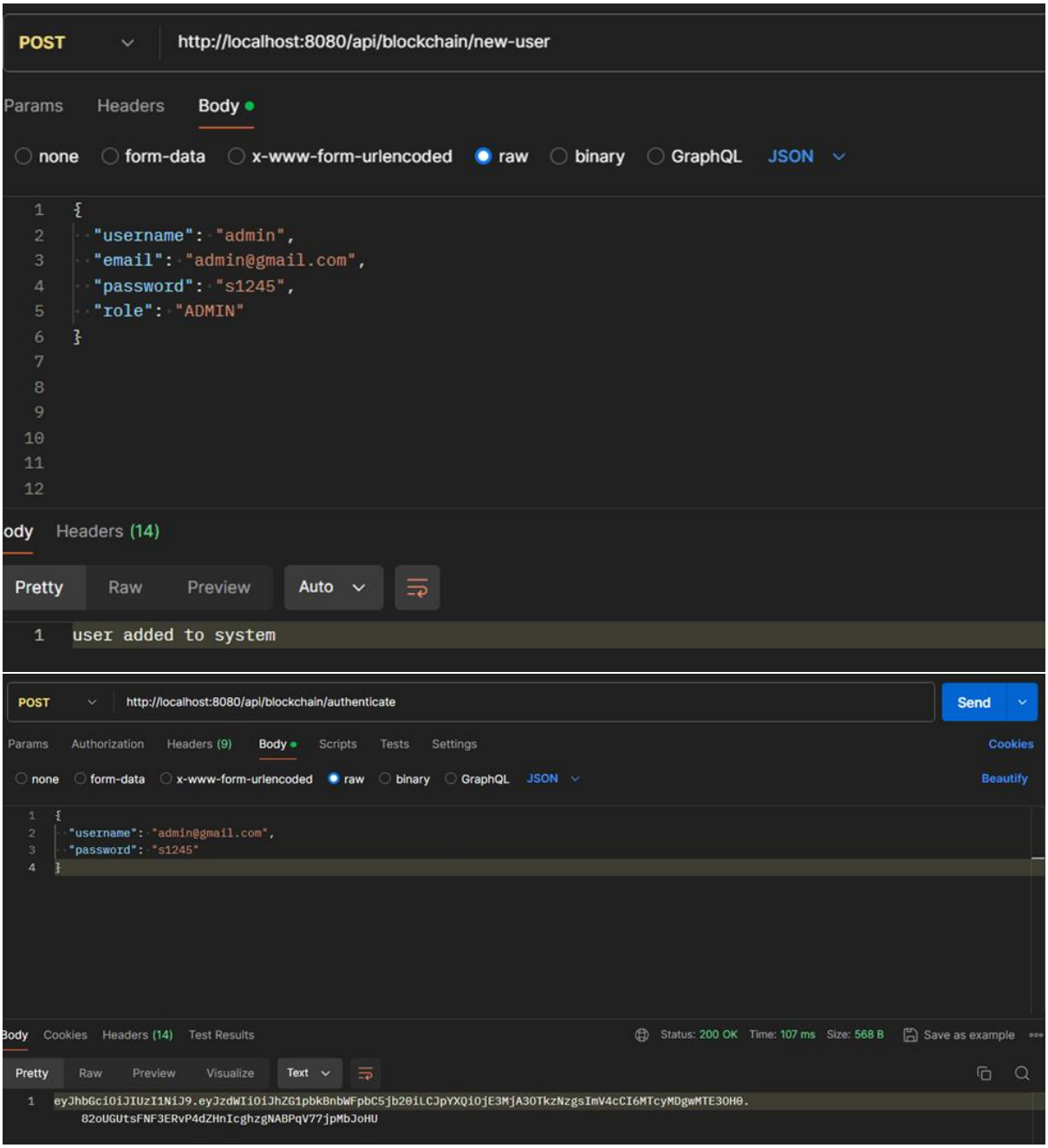


User can access to http://localhost:8080/api/blockchain/full even without JWT because this endpoint is permit for all kind of users.

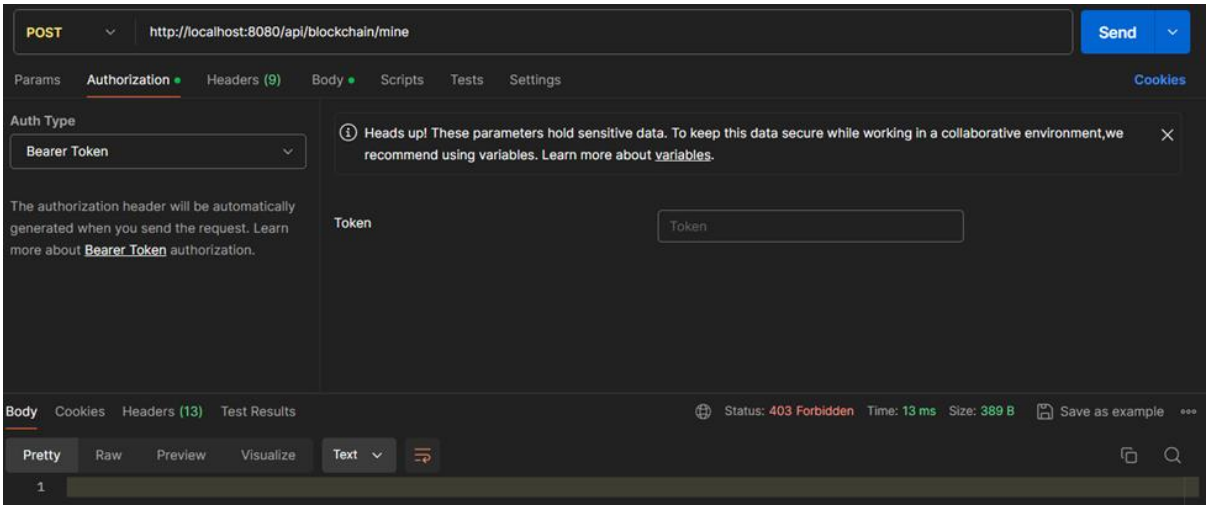


When user try to access http://localhost:8080/api/blockchain/mine endpoint it gives 403 forbidden error, because only ADMIN role is permit to access this endpoint.

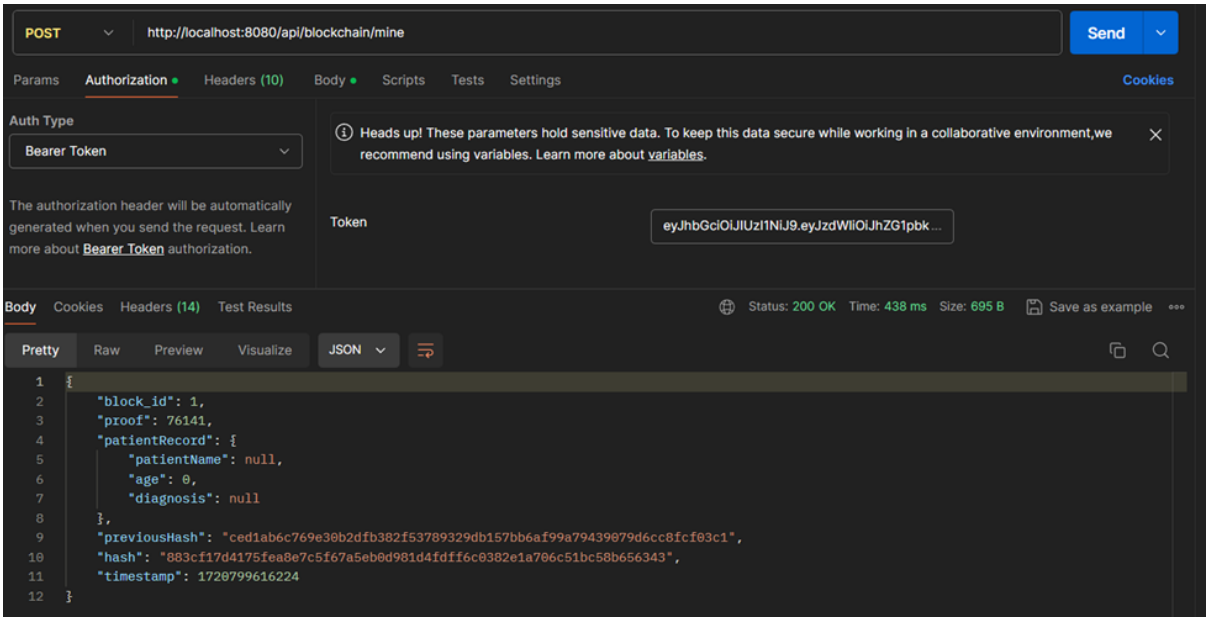
4. Let's create a user as ADMIN. And try to log into system using /authenticate endpoint and receive JWT token.



5. Now try to access http://localhost:8080/api/blockchain/mine endpoint without generated JWT token. Then the response should be 403 – forbidden



Now try to access same endpoint using the generated JWT token. Now the access should be successful.



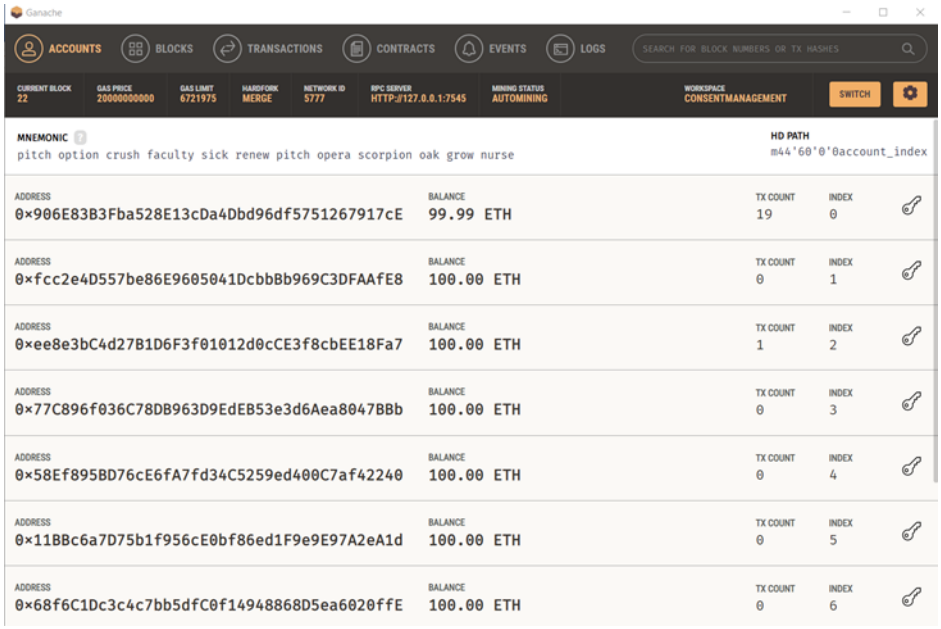
The Role-Based Access Control (RBAC) system in our healthcare blockchain application ensures that only verified users, such as doctors, have access to sensitive patient data, thereby enhancing the security of the Consent Management System. By leveraging Spring Security and JWT authentication, the RBAC system restricts API endpoints based on user roles, preventing unauthorized access. This integration helps maintain stringent access controls within the blockchain network, ensuring that only authorized personnel can request, grant, or manage consent, thereby safeguarding patient data effectively.

6.2.3.3 Consent Management System

In this section, the process by which a patient can accept a consent request via blockchain transactions using smart contracts is demonstrated. This approach ensures a secure, immutable, and transparent method for managing patient consent in a healthcare system.

1. **Setting up the Ganache Test Network**

Ganache is a tool for setting up a local blockchain network for testing Ethereum applications, and its user-friendly interface makes it particularly accessible. To begin, download the Ganache UI from the official Ganache website and install it on your local machine. Once installed, launch Ganache, where you will be presented with options to create a new workspace or use a quickstart Ethereum workspace. For a quick setup, the "Quickstart Ethereum" option is recommended, as it automatically configures and starts a local Ethereum blockchain network. This network comes pre-configured with a list of accounts, each pre-funded with Ether, providing a ready-to-use testing environment.
Connecting to the Ganache network is straightforward, with the default RPC server address being http://127.0.0.1:7545. You can use this address in your Ethereum development tools. Additionally, Ganache provides detailed transaction logs and a block explorer within the UI, aiding in the verification and debugging of smart contracts during development. By using Ganache, developers can ensure their smart contracts are thoroughly tested in a controlled environment before deploying them to a live network, significantly reducing the risk of errors and issues in production.



Ganache UI

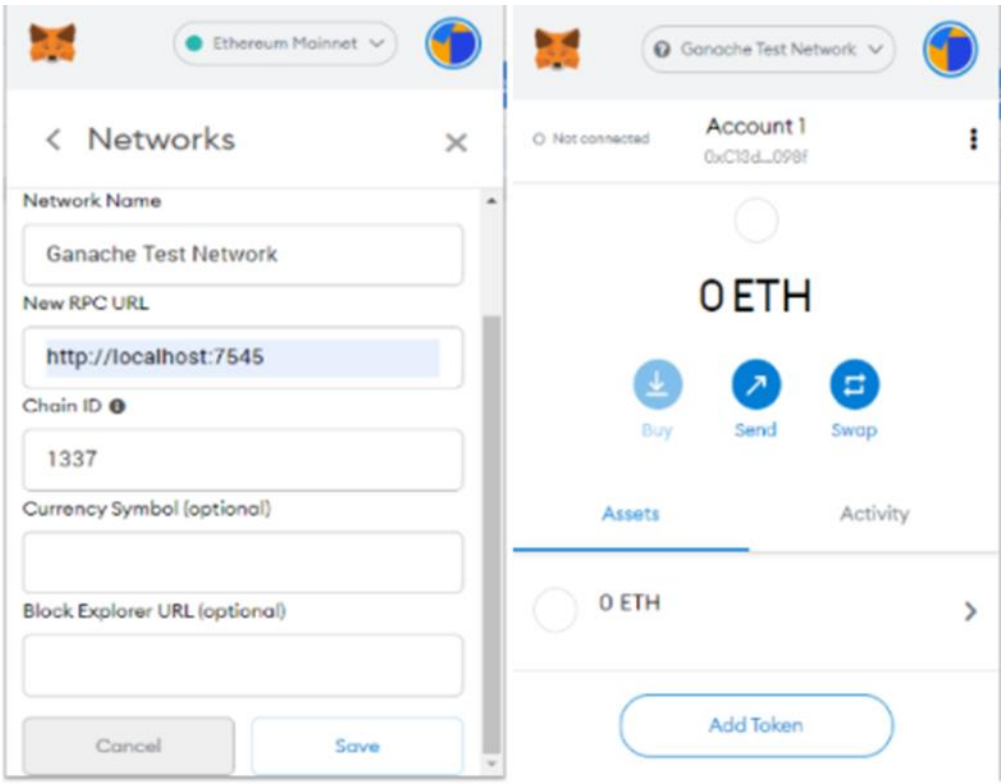
When using the Ganache UI, you'll see a screen displaying various addresses along with their simulated Ether balances. Ganache's advanced settings enable further customization, allowing you to adjust gas limits, block times, and other network parameters to simulate various network conditions. Ganache enables you to run a virtual Ethereum node, which can be connected to a web browser wallet like MetaMask.

2. Connecting Metamask to Ganache

MetaMask is a browser extension designed for connecting to remote Ethereum nodes and facilitating Ethereum wallet management directly through the browser. Upon installation, users are prompted to accept terms and conditions before creating a password. This password serves

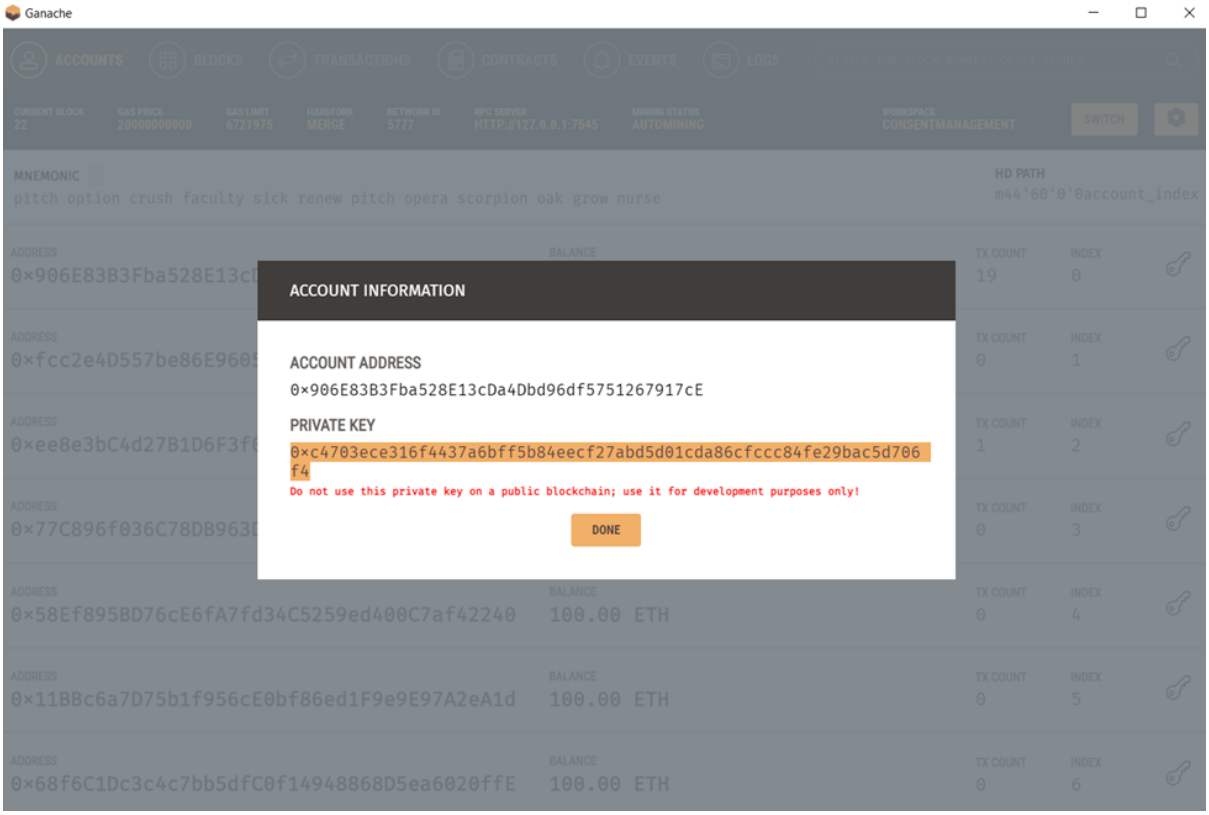
to encrypt the wallets stored in MetaMask and is essential for accessing the extension each time.

After setting your password in MetaMask, you will receive a 12-word seed phrase that acts as a backup to recover your account. It is crucial to securely store this seed phrase since it provides access to your wallet in case you forget your password. Once the setup is complete, you can proceed to connect MetaMask with Ganache. By default, Ganache uses the IP address 127.0.0.1 and port 7545 (localhost:7545), which MetaMask uses to establish a connection with Ganache.

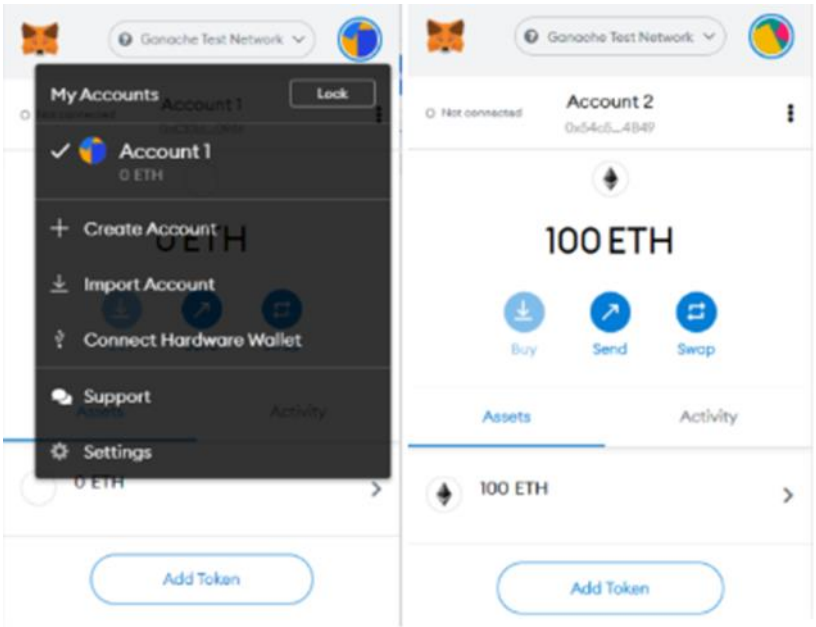


Connecting MetaMask to Ganache

Once connected to the Ganache test network, you can integrate the simulated addresses from Ganache into MetaMask by importing their corresponding private keys. Simply copy the private key from Ganache and paste it into MetaMask under the "Import Account" option. This process ensures that your MetaMask setup aligns with the addresses and simulated Ether available on the Ganache test network, enabling seamless testing and interaction within the Ethereum ecosystem.



The private key of the simulated address



Importing an account to MetaMask from Ganache

6.2.3.4 *Solidity Smart Contract Creation*

Creating smart contracts using Solidity, is crucial for developing decentralized applications and automating transactions on the blockchain. Solidity allows developers to define rules and logic within self-executing contracts, ensuring transparency and security through decentralized execution. Developers write Solidity contracts to deploy on the Ethereum blockchain, defining

contract structures, functions for data manipulation, and business logic for autonomous operations.

Tools like Remix IDE streamline Solidity development by providing syntax highlighting, auto-completion, and a built-in compiler for efficient contract creation, testing, and debugging. Overall, Solidity empowers developers to build decentralized solutions that operate autonomously based on predefined conditions and interactions on the blockchain.

1. ***Registration Contract***

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import "./RelationshipHistory.sol";

contract Registration {
    RelationshipHistory public relationshipHistoryContract;

    mapping(address => bool) public registeredDoctors;

    event DoctorRegistered(address indexed doctor);
    event RelationshipCreated(address indexed doctor, address indexed patient);

    constructor(address _relationshipHistoryContractAddress) {    infinite gas 293800 gas
        relationshipHistoryContract = RelationshipHistory(_relationshipHistoryContractAddress);
    }

    // Function to register a doctor and create a relationship with the patient
    function registerDoctor(address _doctor, address _patient) public {    infinite gas
        require(!registeredDoctors[_doctor], "Doctor is already registered");

        registeredDoctors[_doctor] = true;
        emit DoctorRegistered(_doctor);

        // Create a relationship with the default status of "Requested"
        relationshipHistoryContract.createRelationship(_doctor, _patient);
        emit RelationshipCreated(_doctor, _patient);
    }
    // Function to check if a doctor is registered
    function isDoctorRegistered(address _doctor) public view returns (bool) {    1627 gas
        return registeredDoctors[_doctor];
    }
}
```

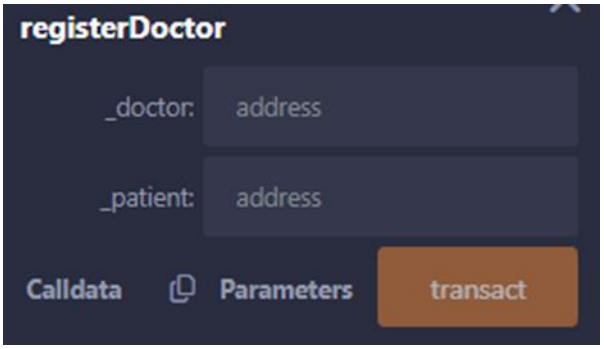
- **State Variables**:
  relationshipHistoryContract: Stores the address of the RelationshipHistory contract, enabling interaction with its functions.

  registeredDoctors: A mapping that tracks whether a doctor is registered (true) or not (false).

- **Constructor**:
  Initializes the relationshipHistoryContract with the provided address of an existing RelationshipHistory contract instance.

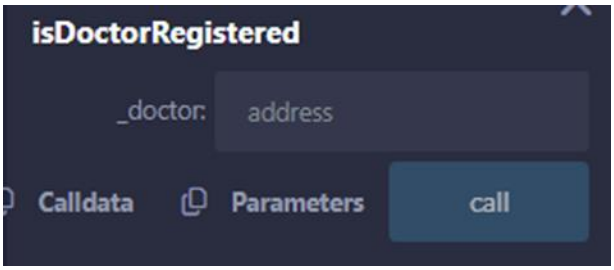
- **Functions**:
  registerDoctor(address _doctor, address _patient):

- Registers a new doctor (_doctor) if not already registered.
- Creates a relationship between the doctor (_doctor) and a patient (_patient) by calling relationshipHistoryContract.createRelationship.
- Emits DoctorRegistered and RelationshipCreated events to log these actions.



isDoctorRegistered(address _doctor):

- Checks if a given doctor (_doctor) is registered by querying the registeredDoctors mapping.
- Returns true if the doctor is registered, otherwise false.



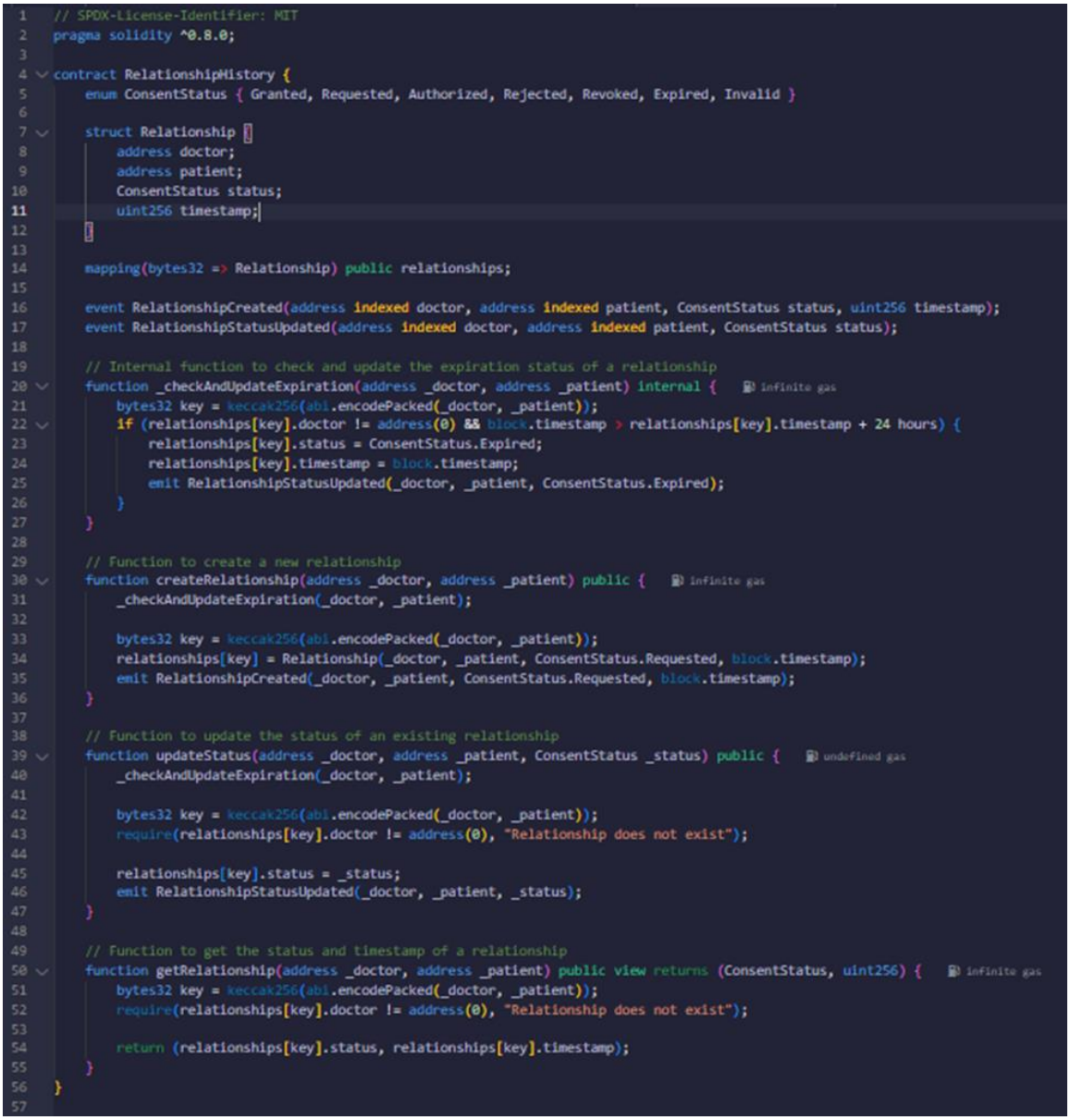
- **Events**:
  DoctorRegistered: Logged when a new doctor is successfully registered.

  RelationshipCreated: Logged when a new relationship between a doctor and a patient is created.

- **Usage**:
  Provides a foundational smart contract for managing doctor registration and patient-doctor relationship creation within a healthcare blockchain application.

  Ensures transparency and auditability through emitted events, allowing external parties to observe and react to changes in doctor registration and patient relationships.

2. ***Relationship History Contract***



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

contract RelationshipHistory {
    enum ConsentStatus { Granted, Requested, Authorized, Rejected, Revoked, Expired, Invalid }

    struct Relationship {
        address doctor;
        address patient;
        ConsentStatus status;
        uint256 timestamp;
    }

    mapping(bytes32 => Relationship) public relationships;

    event RelationshipCreated(address indexed doctor, address indexed patient, ConsentStatus status, uint256 timestamp);
    event RelationshipStatusUpdated(address indexed doctor, address indexed patient, ConsentStatus status);

    // Internal function to check and update the expiration status of a relationship
    function _checkAndUpdateExpiration(address _doctor, address _patient) internal {
        bytes32 key = keccak256(abi.encodePacked(_doctor, _patient));
        if (relationships[key].doctor != address(0) && block.timestamp > relationships[key].timestamp + 24 hours) {
            relationships[key].status = ConsentStatus.Expired;
            relationships[key].timestamp = block.timestamp;
            emit RelationshipStatusUpdated(_doctor, _patient, ConsentStatus.Expired);
        }
    }

    // Function to create a new relationship
    function createRelationship(address _doctor, address _patient) public {
        _checkAndUpdateExpiration(_doctor, _patient);

        bytes32 key = keccak256(abi.encodePacked(_doctor, _patient));
        relationships[key] = Relationship(_doctor, _patient, ConsentStatus.Requested, block.timestamp);
        emit RelationshipCreated(_doctor, _patient, ConsentStatus.Requested, block.timestamp);
    }

    // Function to update the status of an existing relationship
    function updateStatus(address _doctor, address _patient, ConsentStatus _status) public {
        _checkAndUpdateExpiration(_doctor, _patient);

        bytes32 key = keccak256(abi.encodePacked(_doctor, _patient));
        require(relationships[key].doctor != address(0), "Relationship does not exist");

        relationships[key].status = _status;
        emit RelationshipStatusUpdated(_doctor, _patient, _status);
    }

    // Function to get the status and timestamp of a relationship
    function getRelationship(address _doctor, address _patient) public view returns (ConsentStatus, uint256) {
        bytes32 key = keccak256(abi.encodePacked(_doctor, _patient));
        require(relationships[key].doctor != address(0), "Relationship does not exist");

        return (relationships[key].status, relationships[key].timestamp);
    }
}
```

- **State Variables**:
  relationships: A mapping that stores Relationship structs keyed by a hashed combination of doctor and patient addresses.

- **Enums**:
  ConsentStatus: Enumerates the various states a relationship can have, including Granted, Requested, Authorized, Rejected, Revoked, Expired, and Invalid.

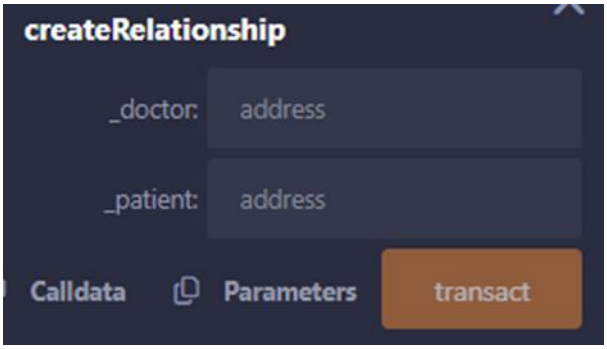
- **Structs**:
  Relationship: Contains details about a relationship, including doctor and patient addresses, status (from ConsentStatus), and timestamp of creation or last status update.

- **Events**:
  RelationshipCreated: Fired when a new relationship is created, capturing details such as the involved doctor and patient, status (typically Requested initially), and timestamp.

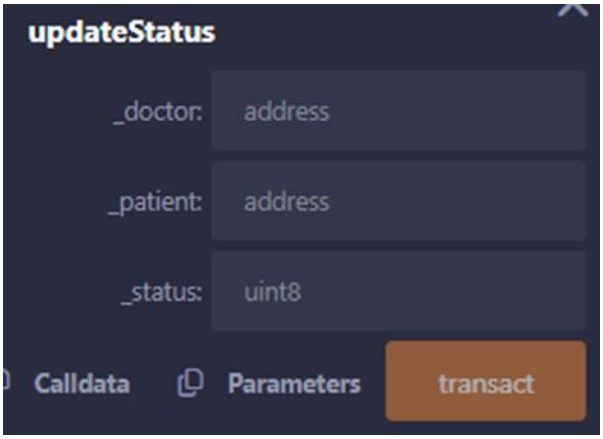
RelationshipStatusUpdated: Triggered when the status of a relationship is updated, indicating changes in consent status (Authorized, Rejected, Revoked, etc.).

- **Functions**:
  _checkAndUpdateExpiration(address _doctor, address _patient): Internal function that verifies if a relationship has expired and updates its status to Expired if necessary.

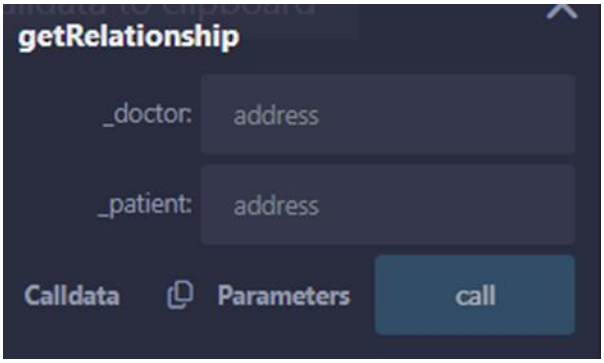
  createRelationship(address _doctor, address _patient): Public function to initiate a new relationship between a doctor and a patient. It sets the initial status as Requested and emits a RelationshipCreated event.



  updateStatus(address _doctor, address _patient, ConsentStatus _status): Public function to update the status of an existing relationship. It ensures the relationship exists and updates its status based on the provided _status. Emits a RelationshipStatusUpdated event.



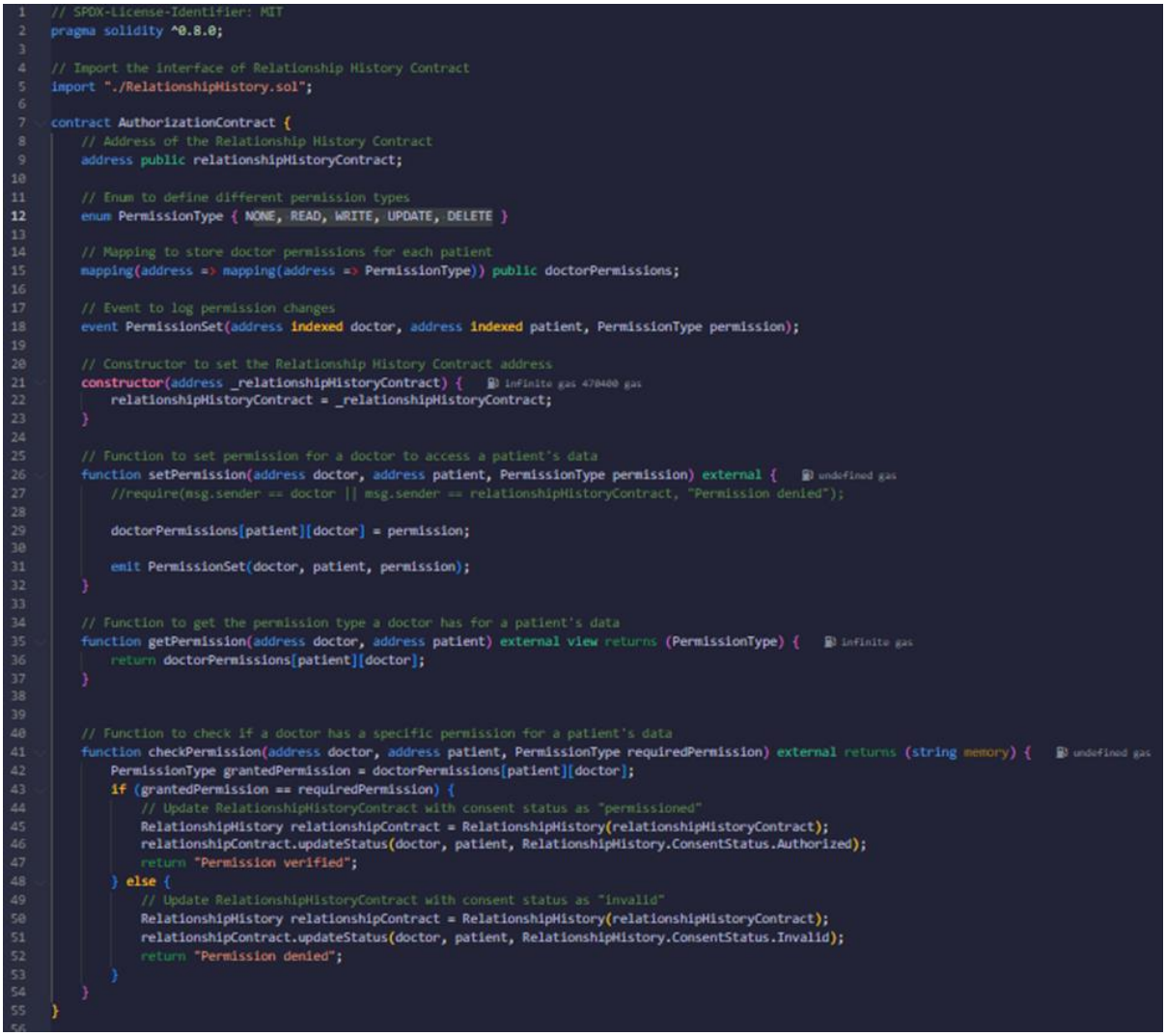
  getRelationship(address _doctor, address _patient): Public view function that retrieves the current status and timestamp of a specified relationship between a doctor and a patient. It validates the existence of the relationship before returning the data.



- **Usage**:
  Provides a robust mechanism to manage consent and authorization statuses between doctors and patients on the blockchain.

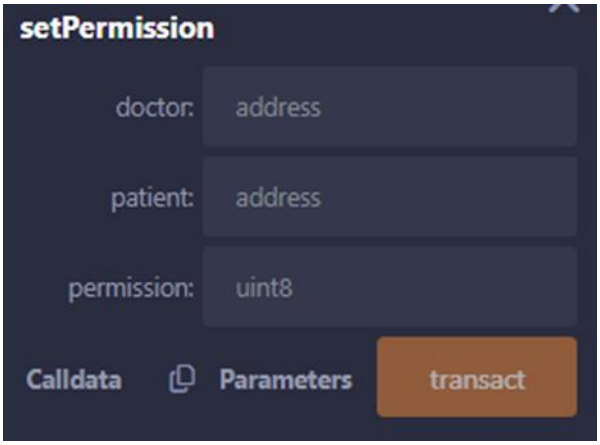
  Ensures transparency and accountability through emitted events, enabling stakeholders to monitor and react to changes in relationship statuses in real-time.

3. ***Authorization Contract***

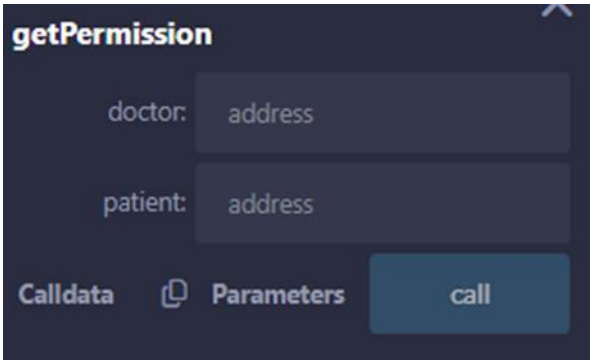


```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

// Import the interface of Relationship History Contract
import "./RelationshipHistory.sol";

contract AuthorizationContract {
    // Address of the Relationship History Contract
    address public relationshipHistoryContract;

    // Enum to define different permission types
    enum PermissionType { NONE, READ, WRITE, UPDATE, DELETE }

    // Mapping to store doctor permissions for each patient
    mapping(address => mapping(address => PermissionType)) public doctorPermissions;

    // Event to log permission changes
    event PermissionSet(address indexed doctor, address indexed patient, PermissionType permission);

    // Constructor to set the Relationship History Contract address
    constructor(address _relationshipHistoryContract) {    infinite gas 470400 gas
        relationshipHistoryContract = _relationshipHistoryContract;
    }

    // Function to set permission for a doctor to access a patient's data
    function setPermission(address doctor, address patient, PermissionType permission) external {    undefined gas
        //require(msg.sender == doctor || msg.sender == relationshipHistoryContract, "Permission denied");

        doctorPermissions[patient][doctor] = permission;

        emit PermissionSet(doctor, patient, permission);
    }

    // Function to get the permission type a doctor has for a patient's data
    function getPermission(address doctor, address patient) external view returns (PermissionType) {    infinite gas
        return doctorPermissions[patient][doctor];
    }


    // Function to check if a doctor has a specific permission for a patient's data
    function checkPermission(address doctor, address patient, PermissionType requiredPermission) external returns (string memory) {    undefined gas
        PermissionType grantedPermission = doctorPermissions[patient][doctor];
        if (grantedPermission == requiredPermission) {
            // Update RelationshipHistoryContract with consent status as "permissioned"
            RelationshipHistory relationshipContract = RelationshipHistory(relationshipHistoryContract);
            relationshipContract.updateStatus(doctor, patient, RelationshipHistory.ConsentStatus.Authorized);
            return "Permission verified";
        } else {
            // Update RelationshipHistoryContract with consent status as "invalid"
            RelationshipHistory relationshipContract = RelationshipHistory(relationshipHistoryContract);
            relationshipContract.updateStatus(doctor, patient, RelationshipHistory.ConsentStatus.Invalid);
            return "Permission denied";
        }
    }
}
```

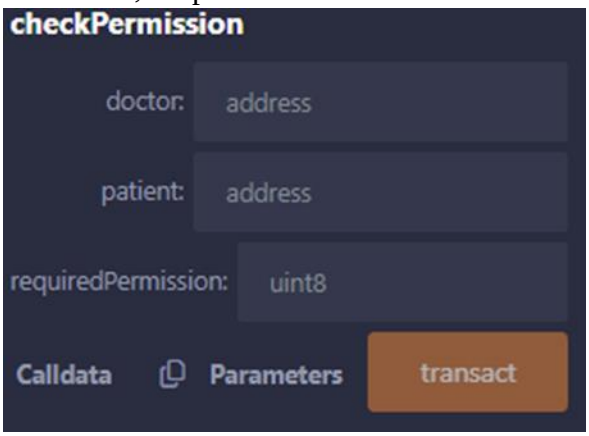
- **State Variables**:
  - relationshipHistoryContract: Stores the address of the RelationshipHistory contract to interact with it.
- **Enums**:
  - PermissionType: Defines permissions granted to doctors for patient data, including NONE, READ, WRITE, UPDATE, and DELETE.
- **Mappings**:
  - doctorPermissions: Nested mapping to store permissions (PermissionType) granted to doctors for each patient.
- **Events**:
  - PermissionSet: Fired when a doctor sets or updates permissions for a patient, capturing details such as the doctor, patient, and the permission granted.
- **Functions**:
  - **Constructor**: Initializes the contract with the address of the RelationshipHistory contract.
  - **setPermission(address doctor, address patient, PermissionType permission)**: Allows a doctor to set or update permissions (permission) for accessing a patient's data (patient). Emits a PermissionSet event upon successful permission setting.



- **getPermission(address doctor, address patient)**: Retrieves the current permission (PermissionType) that a doctor (doctor) has for accessing a specific patient's (patient) data.



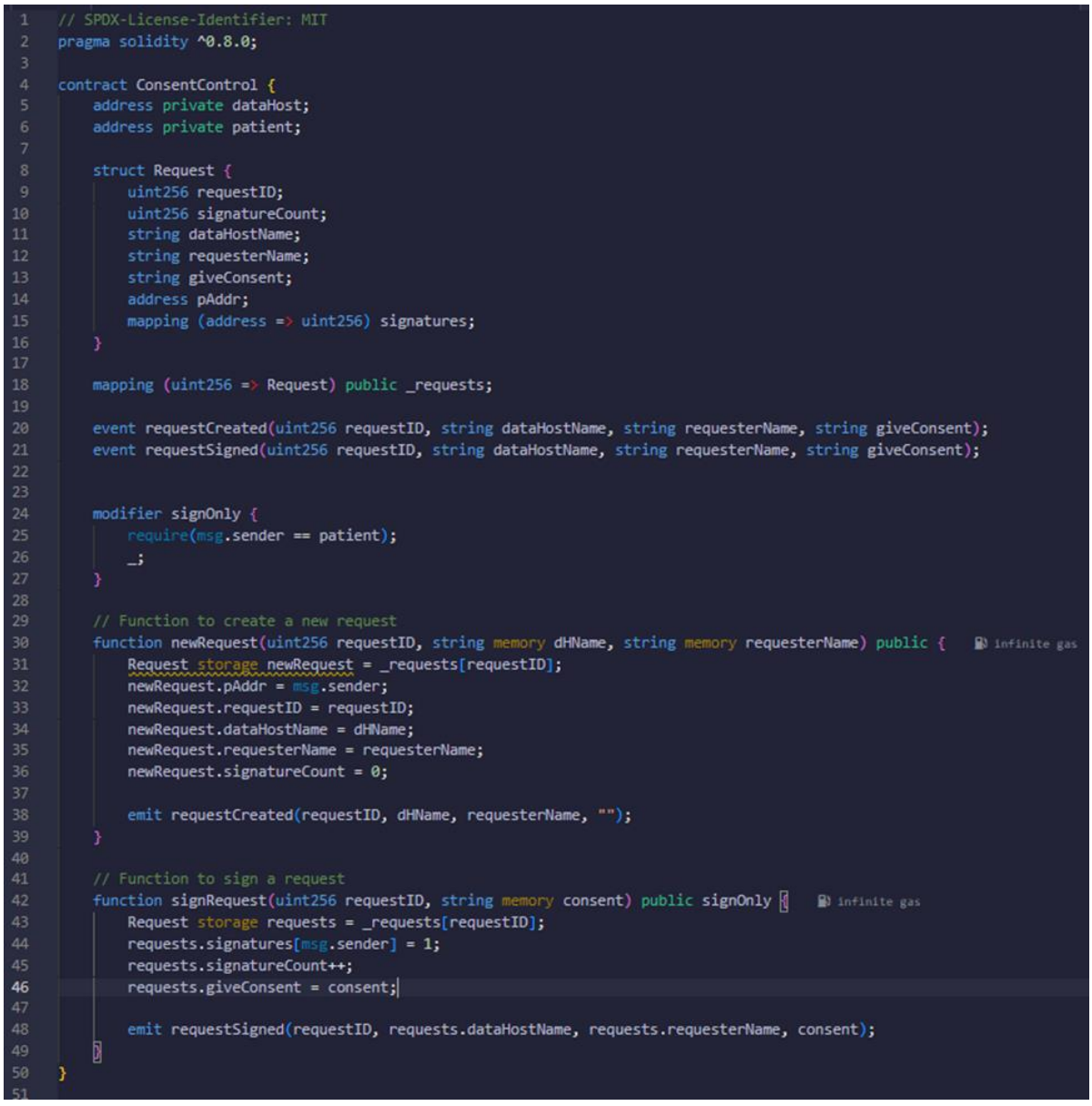
- **checkPermission(address doctor, address patient, PermissionType requiredPermission)**: Verifies if a doctor (doctor) has the required permission (requiredPermission) to access a patient's data (patient). If permissions match, it updates the consent status in the RelationshipHistory contract to Authorized. Otherwise, it updates the status to Invalid and denies access



- **Usage**:
  - Facilitates secure data access by enforcing permissions based on roles (doctor) and data ownership (patient) in the blockchain.

- Ensures compliance with consent management regulations by recording and updating consent statuses (Authorized, Invalid) in the RelationshipHistory contract based on granted permissions.

4. ***Consent Control Contract***

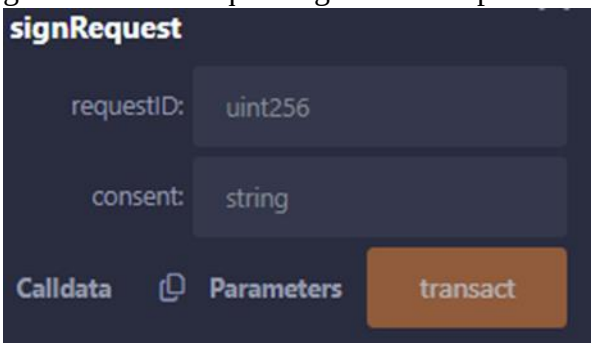


```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ConsentControl {
    address private dataHost;
    address private patient;

    struct Request {
        uint256 requestID;
        uint256 signatureCount;
        string dataHostName;
        string requesterName;
        string giveConsent;
        address pAddr;
        mapping (address => uint256) signatures;
    }

    mapping (uint256 => Request) public _requests;

    event requestCreated(uint256 requestID, string dataHostName, string requesterName, string giveConsent);
    event requestSigned(uint256 requestID, string dataHostName, string requesterName, string giveConsent);


    modifier signOnly {
        require(msg.sender == patient);
        _;
    }

    // Function to create a new request
    function newRequest(uint256 requestID, string memory dHName, string memory requesterName) public {    infinite gas
        Request storage newRequest = _requests[requestID];
        newRequest.pAddr = msg.sender;
        newRequest.requestID = requestID;
        newRequest.dataHostName = dHName;
        newRequest.requesterName = requesterName;
        newRequest.signatureCount = 0;

        emit requestCreated(requestID, dHName, requesterName, "");
    }

    // Function to sign a request
    function signRequest(uint256 requestID, string memory consent) public signOnly {    infinite gas
        Request storage requests = _requests[requestID];
        requests.signatures[msg.sender] = 1;
        requests.signatureCount++;
        requests.giveConsent = consent;

        emit requestSigned(requestID, requests.dataHostName, requests.requesterName, consent);
    }
}
```

- **State Variables**:
  - dataHost: Private variable storing the address of the data host with whom consent requests are managed.
  - patient: Private variable storing the address of the patient who can sign consent requests.
- **Structs**:

    - Request: Defines a structure to represent each consent request, including attributes such as requestID, signatureCount, dataHostName, requesterName, giveConsent, pAddr, and signatures mapping to track signatures from patients.
- **Events**:
    - requestCreated: Triggered when a new consent request is created. Logs the requestID, dataHostName, requesterName, and initial consent status.
    - requestSigned: Fired when a patient signs a consent request. Logs the requestID, dataHostName, requesterName, and the consent given.
- **Modifiers**:
    - signOnly: Modifier restricting access to certain functions to only the patient (msg.sender == patient).
- **Functions**:
    - **newRequest(uint256 requestID, string memory dHName, string memory requesterName)**: Allows the data host to create a new consent request. Initializes a new Request struct and emits a requestCreated event upon creation.

    - **signRequest(uint256 requestID, string memory consent)**: Allows the patient to sign a consent request identified by requestID. Updates the signatures mapping for the patient, increments signatureCount, and records the consent given. Emits a requestSigned event upon successful signing.

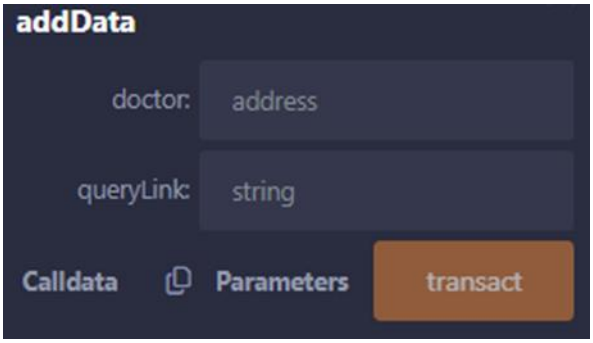


- **Usage**:
    - Enables the controlled initiation and signing of consent requests in a healthcare blockchain system, ensuring transparency and auditability of patient data access permissions.
    - Facilitates compliance with data protection regulations by providing an immutable record of consent transactions (requestCreated and requestSigned events) on the blockchain.

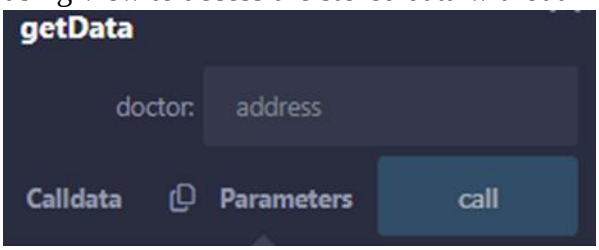
5. ***Data Access Contract***

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

contract DataAccess {
    struct DataEntry {
        string queryLink;
    }

    mapping(address => DataEntry) private doctorData;

    event DataAdded(address doctor, string queryLink);
    event DataRevoked(address doctor);

    function addData(address doctor, string memory queryLink) public {    infinite gas
        doctorData[doctor] = DataEntry(queryLink);
        emit DataAdded(doctor, queryLink);
    }

    function getData(address doctor) public view returns (string memory) {    infinite gas
        DataEntry storage entry = doctorData[doctor];
        return (entry.queryLink);
    }

    function revokeData(address doctor) public {    infinite gas
        delete doctorData[doctor];
        emit DataRevoked(doctor);
    }
}

```

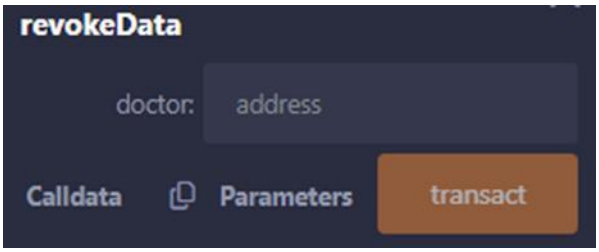
- **State Variables**:
  - doctorData: Private mapping that stores DataEntry structs for each registered doctor, containing a queryLink string.
- **Structs**:
  - DataEntry: Defines a structure holding the queryLink, which represents the link or identifier to access specific data.
- **Events**:
  - DataAdded: Triggered when a doctor adds a new queryLink to their data entry. Logs the doctor address and the queryLink added.
  - DataRevoked: Fired when a doctor revokes their data entry, effectively deleting the queryLink associated with their address.
- **Functions**:
  - **addData(address doctor, string memory queryLink)**: Allows a doctor to add or update their queryLink data entry. Emits a DataAdded event upon successful addition.



- **getData(address doctor) public view returns (string memory)**: Retrieves the queryLink associated with a specific doctor. Operates in a read-only manner using view to access the stored data without modifying the blockchain state.



- **revokeData(address doctor)**: Enables a doctor to revoke their queryLink data entry, deleting the entry from doctorData. Emits a DataRevoked event upon successful revocation.



- **Usage**:
  - Ensures secure and controlled access to specific data entries (queryLink) that doctors registered within the blockchain system.
  - Provides auditability through emitted events (DataAdded, DataRevoked), capturing changes in data access permissions and facilitating compliance with data governance policies.

6.2.4 - Module 04: Facilitating Anonymous Data Provision for Research and Business Analytics

Implementation of Attribute-Based Encryption (ABE) using the Java Pairing-Based Cryptography (JPBC) library in the context of a Spring Boot application for healthcare data on a blockchain.

6.2.4.1 ABEUtil Class

Utility class for ABE operations. This class handles the setup, encryption, and decryption using ABE.

```
package com.example.blockchainhealthcare.utils;

import it.unisa.dia.gas.jpbc.Element;
```

```
import it.unisa.dia.gas.jpbc.Pairing;
import it.unisa.dia.gas.jpbc.PairingParameters;
import it.unisa.dia.gas.plaf.jpbc.pairing.PairingFactory;
import org.springframework.stereotype.Component;

import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;

@Component
public class ABEUtil {

    private Pairing pairing;
    private Element g;
    private Map<String, Element> masterKey;
    private Map<String, Element> publicKey;

    public ABEUtil() {
        try {
            // Load pairing parameters from the params.properties file
            InputStream input = getClass().getResourceAsStream("/params.properties");
            if (input == null) {
                throw new RuntimeException("Failed to find params.properties file");
            }
            PairingParameters params =
PairingFactory.getPairingParameters(input.toString());
            this.pairing = PairingFactory.getPairing(params);
            this.g = pairing.getG1().newRandomElement().getImmutable();
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException("Failed to load pairing parameters", e);
        }
    }

    public Map<String, Element> setup() {
        Element alpha = pairing.getZr().newRandomElement().getImmutable();
        Element gAlpha = g.powZn(alpha);

        publicKey = new HashMap<>();
        publicKey.put("g", g);
        publicKey.put("gAlpha", gAlpha);

        masterKey = new HashMap<>();
        masterKey.put("alpha", alpha);

        return publicKey;
    }

    public Element encrypt(Element publicKey, String data) {
        // Encryption logic using ABE and the given policy
        Element s = pairing.getZr().newRandomElement().getImmutable();
        Element gS = g.powZn(s);



        Element dataElement = pairing.getG1().newElementFromHash(data.getBytes(), 0,
data.length());

        // Ciphertext = (g^s, data * (publicKey^s))
        Element ciphertext = dataElement.mul(publicKey.powZn(s));
```

```
        return ciphertext;
    }

    public Element decrypt(Element privateKey, Element ciphertext) {
        // Decryption logic using ABE and the given attributes


        Element decryptedText = ciphertext.div(privateKey);

        return decryptedText;
    }

}
```

In Patient service handles patient data operations and encrypts patient data before storing it on the blockchain. It is managing patient data and encrypting it before saving to the blockchain.

```
@Override
public void createPatient(PatientDTO patient) throws Exception {
    try {
        // Creating a new patient in the DB
        if (patient.getName().isEmpty() ||
                patient.getDateOfBirth() == null ||
                patient.getAddress().isEmpty() ||
                patient.getContactNumber().isEmpty()) {
            throw new Exception("Please enter all the fields!");
        }
        patient.setCreatedAt(new Date(System.currentTimeMillis()));
        patient.setUpdatedAt(new Date(System.currentTimeMillis()));
        patientRepo.save(patient);

        // Encrypt patient data
        String patientData = patient.toString();
        Map<String, Element> publicKey = abeUtil.setup();
        Element encryptedData = abeUtil.encrypt(publicKey.get("gAlpha"), patientData);

        // Adding a block to the blockchain on POST request
        BlockDTO block =
blockchainUtils.mineBlock(BlockchainEnums.HTTPMethodType.POST,
BlockchainEnums.CollectionType.PATIENTS, patient.getPatientID(),
encryptedData.toString());
        blockchainRepo.save(block);

    } catch (Exception e) {
        throw new Exception(e.getMessage());
    }
}
```

```
@Override
public void updatePatient(String id, PatientDTO patient) throws Exception {
    // Updating a patient in the DB
    Optional<PatientDTO> patientOptional = patientRepo.findById(id);

    if (patient.getName().isEmpty() ||
            patient.getDateOfBirth() == null ||
            patient.getAddress().isEmpty() ||
            patient.getContactNumber().isEmpty()) {
        throw new Exception("Please enter all the fields!");
```

```
    }

    if (patientOptional.isEmpty()) {
        throw new Exception("Patient not found!");
    } else {
        PatientDTO patientToSave = patientOptional.get();

        // Name
        patientToSave.setName(patient.getName());
        // Date of Birth
        patientToSave.setDateOfBirth(patient.getDateOfBirth());
        // Gender
        patientToSave.setGender(patient.getGender());
        // Address
        patientToSave.setAddress(patient.getAddress());
        // Contact Number
        patientToSave.setContactNumber(patient.getContactNumber());
        // Email
        patientToSave.setEmail(patient.getEmail());
        // Insurance Agency ID
        patientToSave.setInsuranceAgencyID(patient.getInsuranceAgencyID());
        // Updated At
        patientToSave.setUpdatedAt(new Date(System.currentTimeMillis()));

        patientRepo.save(patientToSave);

        // Encrypt patient data
        String patientData = patientToSave.toString();
        Map<String, Element> publicKey = abeUtil.setup();
        Element encryptedData = abeUtil.encrypt(publicKey.get("gAlpha"), patientData);

        // Adding a block to the blockchain on PUT request
        BlockDTO block = blockchainUtils.mineBlock(BlockchainEnums.HTTPMethodType.PUT,
BlockchainEnums.CollectionType.PATIENTS, id, encryptedData.toString());
        blockchainRepo.save(block);
    }
}
```

```
@Override
public void deletePatient(String id) throws Exception {
    Optional<PatientDTO> patientOptional = patientRepo.findById(id);

    if (patientOptional.isEmpty()) {
        throw new Exception("Patient not found!");
    } else {
        patientRepo.deleteById(id);

        // Encrypt patient data
        String patientData = ""; // For deletion, we might just store a reference or
empty data
        Map<String, Element> publicKey = abeUtil.setup();
        Element encryptedData = abeUtil.encrypt(publicKey.get("gAlpha"), patientData);

        // Updating the blockchain on DELETE request
        BlockDTO block =
blockchainUtils.mineBlock(BlockchainEnums.HTTPMethodType.DELETE,
BlockchainEnums.CollectionType.PATIENTS, id, encryptedData.toString());
        blockchainRepo.save(block);
```

```
    }
}
```

Method to generate a user's private key and to decrypt data using this key.

```
public PatientDTO decryptPatientData(String encryptedData, String attribute) throws
Exception {
    try {
        // Generate the private key based on the attribute
        Map<String, Element> publicKey = abeUtil.setup();
        Map<String, Element> masterKey = abeUtil.getMasterKey();
        Element privateKey = abeUtil.generatePrivateKey(publicKey, masterKey,
attribute);

        // Decrypt the data
        Element encryptedElement =
abeUtil.getPairing().getG1().newElementFromBytes(encryptedData.getBytes());
        Element decryptedElement = abeUtil.decrypt(privateKey, encryptedElement);

        String decryptedData = new String(decryptedElement.toBytes());



        ObjectMapper objectMapper = new ObjectMapper();
        return objectMapper.readValue(decryptedData, PatientDTO.class);

    } catch (Exception e) {
        throw new Exception("Failed to decrypt patient data", e);
    }
}
```

6.3 - Summary

This chapter discussed the implementation of the novel blockchain model to securely store patient data. The current progress and the current implementations under each module are highlighted here. The next chapter will discuss the opportunities and future prospects and the challenges and implications associated with the usage of blockchain technology in the healthcare sector.

CHAPTER 07 : DISCUSSION

7.1 - Introduction

This chapter mainly discusses the opportunities and future prospects behind the application of blockchain technology in the healthcare sector. Furthermore, it also discusses potential challenges and implications that could be encountered when adopting this disruptive technology in the domain of healthcare.

7.2 - Opportunities & Future prospects

Numerous potential advantages emerge from investigating the possibilities for using blockchain technology in healthcare. The development of conceptual frameworks has a major potential. As healthcare concepts are being developed, researchers are working hard to create safe communication networks and improve data security through algorithmic refinement. As demonstrated by the link between telehealth and communications networks, the development of blockchain-based smart ecosystems for healthcare creates opportunities for more effective and integrated healthcare services.

Blockchain architecture's technological developments highlight even more of the potential for growth. The potential of blockchain technology to improve healthcare operations is demonstrated by the creation of smart ecosystems, technological advancements, and the incorporation of predictive skills. With the help of cloud computing and the Internet of Things, blockchain technology is becoming more and more integrated into AI and healthcare. This has promising implications for diagnostics and predictive medical information.

7.3 - Challenges and Implications

Notwithstanding the enormous potential, it is essential to discuss the difficulties and ramifications of implementing blockchain technology in the healthcare industry. The development of ideas is one of the key issues since blockchain technology has to be improved and tailored to different healthcare fields. It is crucial to protect patient privacy and data security while creating and deploying blockchain-based healthcare systems. Scholars have brought attention to difficulties with memory load, use, and overheating, highlighting the necessity for all-encompassing solutions.

With an emphasis on information handling, privacy, and protection, data management presents a different set of difficulties. Robust authentication techniques and ongoing watchfulness against external attacks are necessary to protect data from unauthorized access and to maintain confidentiality. Furthermore, interoperability, scalability, and overall system performance must be taken into consideration in order to increase the effectiveness of healthcare systems through blockchain integration.

These difficulties have consequences that go beyond technical ones. Healthcare stakeholders are faced with navigating ethical and regulatory issues while making sure standards and rules are followed. Patients and healthcare providers are burdened by rising healthcare expenses as a result of stakeholders' reluctance to cooperate and collaborate for the exchange of health information because of privacy concerns.

7.4 - Evaluation Metrics

7.4.1 - Module 01: Develop a Blockchain Model to Securely Store Medical Data and Propose a Novel Algorithm (Cross-Hash Validator Algorithm) to Enhance the Security of the Stored Data

Module 01 primary focusses on developing the architecture of the blockchain platform. Furthermore, it also introduces a novel algorithm named Cross-Hash Validator algorithm to hash and validate the integrity of the data stored in a database with the association of the blockchain.

To assess the effectiveness and reliability of the CrossHashValidator Algorithm, several evaluation metrics can be employed. These metrics will help determine the algorithm's performance in ensuring data integrity and overall functionality within the blockchain framework. The following are key evaluation metrics:

7.4.1.1 Data Integrity Validation Rate

- **Description**: This algorithm Measures the algorithm's ability to correctly validate data integrity across the blockchain.
- **Calculation**:

$$\frac{\text{Number of Successful Validations}}{\text{Total Number of Validations}} \text{ x } 100$$

- **Importance**: A high validation rate indicates the effectiveness of the algorithm in detecting compromised data.

7.4.1.2 Performance Efficiency

- **Description**: Assesses the time taken to execute the hash and validation processes.
- **Metrics**:
    - **Hashing Time**: Time taken to compute hashes for data records.
    - **Validation Time**: Time taken to perform both vertical and horizontal validation.
- **Importance**: Efficient performance is crucial for real-time applications, particularly in healthcare scenarios where timely access to data is essential.

7.4.1.3 Scalability

- **Description**: Evaluates how the algorithm performs as the size of the blockchain and data records increases.
- **Metrics**:
    - **Time Complexity**: Analyze how execution time increases with the number of blocks and records.
    - **Space Complexity**: Evaluate the memory usage as the dataset scales.
- **Importance**: Ensures that the algorithm remains effective even with large datasets typical in healthcare applications.

7.4.1.4 False Positive Rate

- **Description**: Assesses the frequency of incorrectly identifying valid data as compromised.
- **Calculation**:

$$\frac{\text{Number of False Positives}}{\text{Total Valid Data}} \text{ x } 100$$

- **Importance**: A low false positive rate is crucial to minimize unnecessary alarms and maintain system usability.

7.4.1.5 Conclusion

Utilizing these evaluation metrics will provide a comprehensive assessment of the CrossHashValidator Algorithm's effectiveness, performance, and reliability in maintaining data integrity within a blockchain-based healthcare system. Regular evaluation against these metrics can inform ongoing improvements and adaptations of the algorithm to meet evolving needs and challenges.

7.4.2 - Module 02:

7.4.2.1 Consensus algorithm evaluation criteria

With the rapid development of blockchain technology and its applications in a variety of industries, numerous complicated consensus algorithms have developed, each with its own set of features and functions. The primary purpose of this work is to determine the essential factors influencing the performance of these algorithms. To do this, we did a thorough analysis of the literature and found distinct criteria that are employed in different scenarios, as illustrated in Figure 43. To select the most essential criteria, we employed the pairwise comparison method, which compares each criterion to establish its value or weight. This method is based on the hierarchical decision-making strategy proposed by Saaty (2008) and expanded upon by Odu (2019) and Pamucar, Stevic, and Sremac (2018).[36]

Authors	Year	Performance evaluation criteria
(Croman et al., 2016)	2016	1- Maximum throughput, 2- Latency, 3-Bootstrap time, 4-Cost per Confirmed Transaction, 5-Transaction validation, 6-Bandwidth, 7-Storage
(Baliga, 2017)	2017	1-Transaction finality, 2-Transaction rate, 3-Token needed, 4-Cost of participation, 5-Scalability of the peer network, 6-Trust model, 7-Adversary Tolerance
(Mingxiao et al., 2017b)	2017	1-Byzantine fault tolerance, 2-Crash fault tolerance, 3-Verification speed, 4-Throughput (TPS), 5-Scalability
(Xu, Luthra, Cole, & Blakely, 2018)	2018	1-Architecture (Accounts, Transactions, and Contracts, State Management, Execution Environment), 2-Fault tolerance, 3- Economic Systems Analysis, 4-Block Size, 5-Block Time, 6-Transactional Throughput, 7-Block Throughput, 8-CPU Usage, 9-Transaction Size
(Nguyen & Kim, 2018)	2018	1-Energy efficiency, 2-Modern hardware, 3-Forking, 4-Double spending attack, 5-Block creating speed, 6-Pool mining
(Wang et al., 2018a)	2018	1-Origin of Hardness, 2-Implementation description, 3-ZKP Properties, 4-Simulation of random function, 5-Features of puzzle design, 6-Virtual mining, 7-Simulating Leader election
(Tang et al., 2019)	2019	1-Basic technology, 2-Applicability, 3-TPS, 4-Market capitalization, 5-Number of forks, 6-Total commits in GitHub, 7-Ranking in GitHub, 8-Team activity
(Alsunaidi & Alhaidari, 2019)	2019	1-Node Identity management, 2-Data Model, 3-Electing miners method, 4-Energy saving, 5-Tolerated power of the adversary, 6-Transaction fees, 7-Block reward, 8-Verification speed, 9-Throughput, 10-Block creation speed, 11-Scalability, 12-Extendible 13–51% Attack, 14-Double Spending, 15-Crash Fault Tolerance, 16- Byzantine fault tolerance
(Hasanova et al., 2019)	2019	1-Double spending attack, 2–51% attack, 3-Private key security,4-Noting at stake, 5-criminal problem, 6-selfish mining, 7-block withholding, 8-Bribery attack, 9-DDos/DoS, 10-Sybil attack, 11-Routing attack, 12-Time jacking attack
(Bano et al., 2019)	2019	1-Committee configuration, 2-Transaction censorship resistance, 3-DoS resistance, 4-Adversary model, 5-Throughput, 6-Scalable, 7-Latency, 8-Experimental setup

Figure 43 Summary of different sets of performance evaluation criteria used in blockchain consensus literature[36]

Evaluation Metrics

1.Throughput

- Transactions Per Second (TPS): Measures the number of transactions the network can process per second.
- Block Time/Latency: The time taken to create a new block.
- Block Verification Time: The time required to verify the validity of a block.
- Block Size: The size of each block, which affects the number of transactions it can contain.

2.Profitability of Mining

- Mining Rewards: The incentives given to miners for validating transactions and creating new blocks.
- Power Consumption: The energy required to perform mining operations.
- Transaction Fees: Fees collected from transactions included in a block.
- Special Hardware Dependency: The need for specialized hardware to perform mining.

3.Decentralization Levels

- Governance: The mechanism through which decisions are made within the blockchain network.
- Permission Model: The level of access control over who can participate in the consensus process.
- Trust Model: The method used to establish trust among the participants in the network.

4.Consensus Algorithms Vulnerabilities

- Double Spending Attack: The risk of spending the same digital token more than once.

- 51% Attack: The possibility of a single entity controlling the majority of the network's computational power.
- Sybil Attack: The creation of multiple fake identities to gain influence within the network.
- Routing Attack: Interfering with the network communication to isolate or delay nodes.
- Time Jacking Attack: Manipulating the network time to disrupt the consensus process.

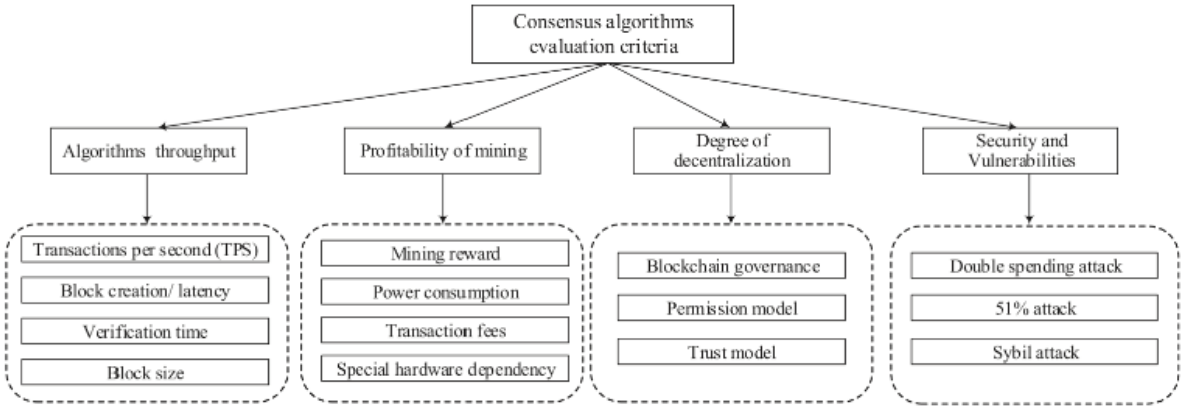


Figure 44 A performance evaluation framework for blockchain consensus algorithms.[36]

Consensus algorithms	Designing Goal	Decentralization level	Permission model/ Node Identity Management	Electing Miners/ verifiers Based on	Energy efficiency	Scalability	%51 Attack	Double Spendingattack	Hardware dependency	speed
PoW	Sybil-proof	Decentralized	Permissionless	Work (Hash)	No	Strong	Vulnerable	Vulnerable	Yes	Slow
PoS	Energy efficiency	Semi-centralized	Permissionless	Stake	Yes	Strong	Vulnerable	Difficult	No	Fast
DPoS	Organize PoS effectively	Semi-centralized	Both	Vote	Yes	Strong	Vulnerable	Vulnerable	No	Fast
PBFT	Remove software errors	Decentralized	Both	Vote	Yes	Low	Safe	Safe	No	Slow
PoC	Less energy than PoW	Decentralized	Permissionless	Work (Hash)	Fair	Strong	Vulnerable	Vulnerable	Yes	Slow
DAG	Speed and Scalability	Decentralized	Permissionless	N/A	Yes	Strong	Safe	Safe	No	Fast
PoA	Benefits of both Pos and PoW	Decentralized	Permissioned	Vote and work	No	Strong	Safe	Vulnerable	Yes	Fair
dBFT	Faster PBFT	Semi-centralized	Permissioned	Vote	Yes	Medium	Vulnerable	Vulnerable	No	Slow
PoI	Improve PoS	Decentralized	Permissionless	Importance scores	Yes	Strong	Safe	Safe	No	Fast
PoB	N/A	Decentralized	Permissionless	Burnt coins	No	Medium	Vulnerable	vulnerable	No	Fast

Figure 45 Comparison of blockchain consensus algorithms [36]

Proposed custom consensus algorithm comparison.

Consensus algorithm	Designing Goal	Decentralization level	Permission model	Electing Miners	Energy Efficiency	Scalability	%51 Attack	Double Spending attack	Hardware Dependency	speed
PoAV (Improved PoA)	Speed, less energy and Secured	Decentralized	Permissioned	Vote	Yes	Strong	Safe	Safe	No	Fast
PoA	Speed and Less energy	Centralized	Permissioned	N\A	Yes	Strong	Safe	Safe	No	Fast

7.4.3.1 Evaluation

The evaluation of the proposed Consent Management System (CMS) focuses on assessing its effectiveness, security, and compliance in managing consent within a decentralized environment. This evaluation is based on the CMS design, its features, functionalities, and the integration of innovative ideas for a new design. Additionally, it includes a comprehensive comparison between existing designs and the proposed design, identifying the strengths of the new approach

By thoroughly evaluating these aspects, this section aims to provide a comprehensive understanding of the CMS's strengths and areas for development, ensuring it meets the needs of managing consent in the digital age effectively and securely. The evaluation considers how the proposed design leverages blockchain technology and smart contracts to enhance transparency, traceability, and patient control over health data. It also examines the system's compliance with GDPR requirements, its security mechanisms, and its potential to improve over existing consent management solutions. This thorough assessment will highlight the unique advantages of the proposed CMS and provide insights for future enhancements and implementations.

1. **Utilizing Series of Smart Contracts For CMS**

In an era marked by extensive digital interactions and the widespread sharing of personal data, the demand for robust and comprehensive consent management systems has grown significantly essential. Traditionally, a single smart contract has been utilized to manage consent, but recent research suggests that a series of interconnected smart contracts can provide a more effective and flexible approach.

Blockchain technology has emerged as a promising solution for consent management, as it offers decentralized, secure, and transparent data management capabilities. Particularly, the implementation of smart contracts on a blockchain can automate the negotiation, verification, and enforcement of consent agreements.

However, existing approaches to blockchain-based consent management systems have faced certain limitations. To address these challenges, the utilization of a series of smart contracts, rather than a single contract, can offer several advantages.

One key benefit of employing a series of smart contracts is the ability to divide the consent management process into distinct, modular components. This allows for greater flexibility, scalability, and maintainability of the system. By breaking down the consent management process into smaller, specialized contracts, the system can more efficiently handle complex consent scenarios, such as dynamic consent preferences, consent revocation, and consent delegation.

Another advantage of using interconnected smart contracts is enhanced security and privacy. By segregating different aspects of consent management into separate contracts, sensitive data can be compartmentalized and access rights can be more finely controlled. This segregation reduces the risk of unauthorized access or manipulation of consent-related information, thereby bolstering data security and compliance with privacy regulations like GDPR.

Moreover, a modular approach with multiple smart contracts allows for better auditability and transparency. Each contract can be designed to log specific actions and events related to consent, providing a clear audit trail of how consent agreements are negotiated, updated, and enforced over time. This transparency not only enhances accountability within the system but also builds trust among related parties in the CMS.

Furthermore, the use of interconnected smart contracts facilitates interoperability with other systems and platforms. Different components of the consent management process, such as identity verification, data access controls, and consent validation, can be integrated with existing healthcare infrastructures or third-party services more seamlessly. This interoperability enhances the overall efficiency of consent management systems, enabling smoother data exchanges while maintaining compliance with regulatory requirements.

In conclusion, adopting a series of interconnected smart contracts for consent management represents a forward-thinking approach to leveraging blockchain technology in healthcare and beyond. By addressing the limitations of traditional single-contract systems, this approach offers enhanced flexibility, security, auditability, and interoperability, thereby supporting more robust and compliant consent management solutions.

2. Fulfilling GDPR Requirements

Ensuring compliance with GDPR regulations is crucial as healthcare integrates advanced technologies like Blockchain. Instances of patient data misuse and breaches highlight the risks associated with adopting new technologies, potentially eroding trust in healthcare institutions and hindering the adoption of beneficial innovations. GDPR imposes stringent requirements on data processing in Blockchain applications within healthcare. Chapter 3, Table 2 outlines key requirements essential for leveraging Blockchain in healthcare, focusing primarily on implementing consent management through smart contracts. However, this design also addresses several other GDPR provisions, with some challenges yet to be fully resolved to enable effective Blockchain utilization in healthcare.

Article 6 of GDPR pertains to the lawful basis for processing data, a criterion met in our design. When a healthcare provider or third party like doctor seeks access to patient health data, the request is recorded on the Blockchain. Consent from the patient is mandatory for data processing, fulfilling the first of the six conditions under Article 6 [66]. Article 17, known as the "Right to be forgotten," presents one of the most complex GDPR provisions potentially limiting Blockchain adoption in healthcare. The immutability of Blockchain poses a challenge to compliance, as data once recorded cannot be erased. Acknowledging this challenge, this solution segregates sensitive data, enabling its deletion upon patient request, while storing consent records, query links, and encrypted data on the Blockchain via smart contracts.

This approach ensures transparency and immutability through Blockchain while securely managing patient data off-chain, facilitating compliance with GDPR's stringent data security requirements. Addressing the "Right to be forgotten," patient data deletions are simplified; patients simply request from CMS to delete data, including any shared query links. Additionally, Article 32 of GDPR mandates encryption or pseudonymization of personal data during processing, a requirement met by encrypting data within our off-chain data host.

Article 7 of GDPR, which mandates patient consent, serves as the foundational principle driving this design. Design's primary objective revolves around leveraging Blockchain technology to establish robust consent management for data sharing in eHealth contexts, specifically for managing consents related to access personal health data. Illustrated in Figure 16, this design meticulously outlines the process of consent management when a doctor requests access to patient health records. This framework encompasses accepting requests, securing the patient's signed consent, and facilitating the transfer of requested health data to the requester. Furthermore, Blockchain ensures the integrity and versioning of patient-signed consent requests throughout their lifecycle.

Central to this design is the adherence to GDPR's stipulation that individuals have the right to withdraw consent at any time, as detailed in Section 5.3 of Chapter 5. Our solution ensures that withdrawing consent is as straightforward as granting it. Patients initiate consent revocation requests directly within the Blockchain framework, prompting update corresponding smart contracts. This mechanism guarantees that consent management remains responsive and compliant, aligning with GDPR's stringent requirements for data protection and individual rights in Healthcare scenarios.

In conclusion, while integrating Blockchain into healthcare presents challenges for GDPR compliance, this design leverages Blockchain's strengths to enhance data integrity, streamline consent management, and empower patients with data control. By segregating data and leveraging Blockchain for transparency and immutability while storing sensitive data off-chain, this navigate GDPR complexities, ensuring data security and regulatory compliance in healthcare settings.

3. Two Layer Security by integrating with RBAC

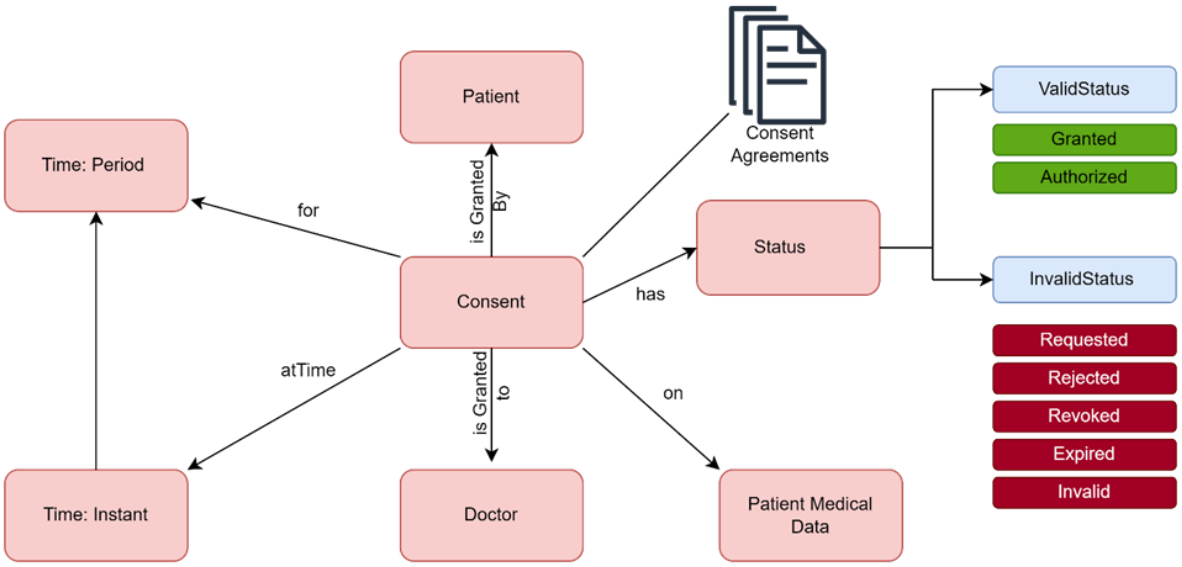
Integrating Role-Based Access Control (RBAC) into consent management systems offers a robust two-layer security approach that significantly strengthens data protection and access management in healthcare settings. RBAC's primary advantage lies in its ability to finely tune access privileges based on predefined roles, ensuring that only authorized personnel, such as healthcare providers or doctors, can access specific patient data. This granularity aligns perfectly with GDPR principles by restricting access to personal health information (PHI) strictly to those who have a legitimate need as defined by their roles within the organization.

Moreover, RBAC enhances overall data security by enforcing strict controls over data access and operations. It empowers consent management system to govern who can view, modify, or delete patient consent data under well-defined conditions. By preventing unauthorized access and mitigating the risks of data breaches or misuse. RBAC acts as a protective shield, safeguarding sensitive consent information by preventing unauthorized access and ensuring compliance with regulatory frameworks like GDPR. By structuring access permissions according to roles and responsibilities, RBAC enhances data security, operational integrity, and regulatory adherence in healthcare environments where maintaining privacy and consent management are paramount.

Furthermore, RBAC frameworks are adaptable and scalable, capable of accommodating evolving organizational needs and changing regulatory landscapes without requiring significant modifications to the consent management infrastructure. This adaptability ensures that healthcare systems can efficiently manage access rights across different user groups, supporting operational efficiency while upholding stringent data protection standards essential in modern healthcare environments.

4. Ontology Representation in Consent Status

The fundamental components of consent are delineated in Figure 1 through ontology representation. These encompass the data subject, specific details of personal data, the identity of the consent requester, evidence of agreement, contextual descriptors (including purposes, timeframe, operations, and scope), and the status of consent, which can either be valid or invalid. Consent is granted directly by the patient to a data requester, constituting crucial attributes for legally valid consent. In scenarios where patient data collection and utilization hinge upon consent, the data controller must be capable of demonstrating that consent has been obtained. The consent agreement must explicitly outline the identities of the consent requester and data controller, the purposes and methods of data collection, usage conditions, and the processing activities entailed. Furthermore, the patient retains the right to withdraw consent at any time, without affecting the legality of data collection and use performed prior to withdrawal.



Consent Concept representation using ontology

7.4.3.2 Discussion

The proposed Consent Management System represents a significant advancement in the handling of consent within the context of data protection and legal frameworks. By leveraging a series of smart contracts, the system ensures transparency, security, and automation in managing consent processes. Compliance with GDPR is seamlessly integrated, ensuring that all regulatory requirements are met. The two-layer security mechanism, combining blockchain technology with RBAC, provides robust protection against both external and internal threats. Finally, the use of ontology representation for consent status ensures that the system can capture the complexity of consent, providing a detailed and accurate view of the consent lifecycle. Together, these features create a sophisticated and reliable solution for consent management, addressing both regulatory requirements and the need for enhanced data security.

7.4.3.3 Limitations

Despite the strengths of the proposed Consent Management System (CMS), several limitations must be addressed to fully realize its potential. One significant limitation is that the system

does not utilize Role-Based Access Control (RBAC) for interactions with smart contracts on the blockchain. While RBAC is employed to secure the off-chain components of the system, such as user access to the consent management interface, it is not extended to manage and restrict transactions that occur on the blockchain itself. This omission means that any user with access to the blockchain can potentially interact with smart contracts, which could expose the system to unauthorized actions. Implementing RBAC for smart contract interactions would add an additional layer of security, ensuring that only authorized entities can execute transactions or query data on the blockchain.

Another limitation involves the handling of medical data, which is stored off-chain due to the current constraints of blockchain technology. While the data is encrypted to ensure privacy and security, smart contracts are not yet capable of decrypting and accessing this data directly. This decryption process must be performed by an external application, such as a Spring Boot application, which adds complexity to the system and introduces potential points of failure. The reliance on off-chain storage and decryption limits the autonomy and self-sufficiency of the smart contracts, potentially complicating the integration and management of the entire system.

The system currently does not employ separate smart contracts to facilitate easy interactions between contracts, such as using eth calls within the Ethereum network. Eth calls allow for executing and receiving messages instantly without creating a standard blockchain transaction, providing a seamless and efficient way to interact with smart contracts. The absence of this mechanism means that each interaction requires a traditional blockchain transaction, which can be slower and more resource-intensive. Incorporating separate contracts designed for eth calls would enhance the system's efficiency and responsiveness, allowing for smoother and more user-friendly interactions.

While the proposed CMS offers a robust framework for managing consent using blockchain technology, addressing these limitations would further enhance its functionality and security. Integrating RBAC for smart contract interactions would strengthen access control on the blockchain, ensuring that only authorized users can perform transactions. Overcoming the dependency on off-chain decryption processes would simplify the system architecture and improve its reliability. Additionally, utilizing separate contracts for seamless interactions via eth calls would enhance the system's efficiency, providing a more streamlined user experience. By addressing these areas, this CMS can become a more comprehensive and resilient solution for consent management.

7.4.4 - Module 04: Evaluation Metrics for Secure and Anonymous Data Provision Module

To evaluate the performance of the proposed system, several key metrics were analyzed. Latency was measured in terms of encryption latency, which is the time taken to encrypt data using CP-ABE; storage latency, which is the time taken to store encrypted data on the blockchain; retrieval latency, which is the time taken to retrieve encrypted data from the blockchain; and decryption latency, which is the time taken to decrypt data using the issued private key. Low latency is crucial for ensuring timely access to medical data, especially in urgent healthcare scenarios.

Scalability was assessed by measuring performance metrics, specifically latency and throughput, under increasing data volumes and numbers of concurrent users. Scalability is

essential for the system to grow and adapt to increasing demands, ensuring its long-term viability in a healthcare environment. The effectiveness of CP-ABE in enforcing fine-grained access control based on user attributes was also evaluated. This was measured by the success rate of authorized access attempts (users with correct attributes) and the failure rate of unauthorized access attempts (users without required attributes). Ensuring that only authorized users can access sensitive medical data is vital for maintaining patient privacy and data security.

7.4.4.1 Latency:

- **Measurement**:
  - Encryption Latency: Time taken to encrypt data using CP-ABE.
  - Storage Latency: Time taken to store encrypted data on the blockchain.
  - Retrieval Latency: Time taken to retrieve encrypted data from the blockchain.
  - Decryption Latency: Time taken to decrypt data using the issued private key.
- **Importance**: Low latency is crucial for ensuring timely access to medical data, especially in urgent healthcare scenarios.

7.4.4.2 Scalability:

- **Measurement**: Performance metrics (latency and throughput) under increasing data volumes and numbers of concurrent users.
- **Importance**: Scalability ensures the system can grow and adapt to increasing demands, which is critical for long-term viability in a healthcare environment.

7.4.4.3 Access Control Effectiveness:

The effectiveness of CP-ABE in enforcing fine-grained access control based on user attributes.

- **Measurement**:
  - Success rate of authorized access attempts (users with correct attributes).
  - Failure rate of unauthorized access attempts (users without required attributes).
- **Importance**: Ensures that only authorized users can access sensitive medical data, maintaining patient privacy and data security.

7.5 - Summary

In conclusion, there are several chances to transform healthcare operations through the application of blockchain technology in the industry. The development of conceptual frameworks, advances in technology, and gains in efficiency point to a bright future for improved healthcare services. However, there are drawbacks to these potential as well, from technological difficulties to moral and legal issues. To fully realize blockchain's promise in healthcare, these issues must be resolved. As the healthcare sector embraces blockchain's revolutionary potential, ongoing research and teamwork are crucial to overcoming obstacles and guaranteeing the technology's smooth integration for improved healthcare services. The intelligent and strategic use of blockchain technology will shape the healthcare landscape of the future and create a safe, open, and effective healthcare system.

CHAPTER 08 : REFERENCES

[1] McClean, S., Gillespie, J., Garg, L., Barton, M., Scotney, B. and Kullerton, K., 2014. Using phase-type models to cost stroke patient care across health, social and community services. *European Journal of Operational Research*, *236*(1), pp.190-199.

[2] Quadery, S.E.U., Hasan, M. and Khan, M.M., 2021. Consumer side economic perception of telemedicine during COVID-19 era: A survey on Bangladesh's perspective. *Informatics in Medicine Unlocked*, *27*, p.100797.

[3] Chanda, J.N., Chowdhury, I.A., Peyaru, M., Barua, S., Islam, M. and Hasan, M., 2021, October. Healthcare Monitoring System for Dedicated COVID-19 Hospitals or Isolation Centers. In *2021 IEEE Mysore Sub Section International Conference (MysuruCon)* (pp. 405-410). IEEE.

[4] Ghosh, P.K., Chakraborty, A., Hasan, M., Rashid, K. and Siddique, A.H., 2023. Blockchain application in healthcare systems: a review. *Systems*, *11*(1), p.38.

[5] Ghayvat, H., Pandya, S., Bhattacharya, P., Zuhair, M., Rashid, M., Hakak, S. and Dev, K., 2021. CP-BDHCA: Blockchain-based Confidentiality-Privacy preserving Big Data scheme for healthcare clouds and applications. *IEEE Journal of Biomedical and Health Informatics*, *26*(5), pp.1937-1948.

[6] E. Andreas, M. Raimundas, and M. Nicolas, 'A Comprehensive Reference Model for Blockchain-based Distributed Ledger Technology', [Online]. Available: https://ceur-ws.org/Vol-1979/paper-09.pdf

[7] Y. Wu, X. Zhang, and H. Sun, 'A multi-time-scale autonomous energy trading framework within distribution networks based on blockchain', Applied Energy, vol. 287, p. 116560, Apr. 2021, doi: 10.1016/j.apenergy.2021.116560.

[8] 'A review on consensus algorithm of blockchain | IEEE Conference Publication | IEEE Xplore'. Accessed: Dec. 30, 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8123011

[9] C. Kombe, M. A. Dida, and A. E. Sam, 'A review on healthcare information systems and consensus protocols in blockchain technology', 2018, Accessed: Dec. 30, 2023. [Online]. Available: https://www.academia.edu/download/88816259/1.pdf

[10] D. Yang, C. Long, H. Xu, and S. Peng, 'A Review on Scalability of Blockchain', in Proceedings of the 2020 The 2nd International Conference on Blockchain Technology, in ICBCT'20. New York, NY, USA: Association for Computing Machinery, May 2020, pp. 1–6. doi: 10.1145/3390566.3391665.

[11] 'A Survey Paper On Consensus Algorithm Of Mobile-Healthcare In Blockchain Network | IEEE Conference Publication | IEEE Xplore'. Accessed: Dec. 30, 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9077737

[12] M. Hölbl, M. Kompara, A. Kamišalić, and L. Nemec Zlatolas, ‘A Systematic Review of the Use of Blockchain in Healthcare’, Symmetry, vol. 10, no. 10, Art. no. 10, Oct. 2018, doi: 10.3390/sym10100470.

[13] A. Chauhan, O. P. Malviya, M. Verma, and T. S. Mor, ‘Blockchain and Scalability’, in 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Jul. 2018, pp. 122–128. doi: 10.1109/QRS-C.2018.00034.

[14] T.-T. Kuo, H.-E. Kim, and L. Ohno-Machado, ‘Blockchain distributed ledger technologies for biomedical and health care applications’, Journal of the American Medical Informatics Association, vol. 24, no. 6, pp. 1211–1220, Nov. 2017, doi: 10.1093/jamia/ocx068.

[15] M. Attaran, ‘Blockchain technology in healthcare: Challenges and opportunities’, International Journal of Healthcare Management, vol. 15, no. 1, pp. 70–83, Jan. 2022, doi: 10.1080/20479700.2020.1843887.

[16] B. K. Mohanta, D. Jena, S. S. Panda, and S. Sobhanayak, ‘Blockchain technology: A survey on applications and security privacy Challenges’, Internet of Things, vol. 8, p. 100107, Dec. 2019, doi: 10.1016/j.iot.2019.100107.

[17] Z. Wenhua, F. Qamar, T.-A. N. Abdali, R. Hassan, S. T. A. Jafri, and Q. N. Nguyen, ‘Blockchain Technology: Security Issues, Healthcare Applications, Challenges and Future Trends’, Electronics, vol. 12, no. 3, Art. no. 3, Jan. 2023, doi: 10.3390/electronics12030546.

[18] I. M. Al-Joboury and E. H. Al-Hemiary, ‘Consensus algorithms based blockchain of things for distributed Healthcare’, Iraqi Journal of Information and communication technology, vol. 3, no. 4, pp. 33–46, 2020.

[19] H. Ruan, H. Gao, H. Qiu, H. B. Gooi, and J. Liu, ‘Distributed operation optimization of active distribution network with P2P electricity trading in blockchain environment’, Applied Energy, vol. 331, p. 120405, Feb. 2023, doi: 10.1016/j.apenergy.2022.120405.

[20] B. He and T. Feng, ‘Encryption Scheme of Verifiable Search Based on Blockchain in Cloud Environment’, Cryptography, vol. 7, no. 2, Art. no. 2, Jun. 2023, doi: 10.3390/cryptography7020016.

[21] T. Frikha, F. Chaabane, N. Aouinti, O. Cheikhrouhou, N. Ben Amor, and A. Kerrouche, ‘Implementation of Blockchain Consensus Algorithm on Embedded Architecture’, Security and Communication Networks, vol. 2021, p. e9918697, Apr. 2021, doi: 10.1155/2021/9918697.

[22] A. Huskanović, ‘Proof of Work - What It Is and How Does It Work’, Async Labs - Software Development & Digital Agency. Accessed: Dec. 30, 2023. [Online]. Available: https://www.asynclabs.co/blog/blockchain-development/proof-of-work-what-it-is-and-how-does-it-work/

[23] M. Pisa, ‘Reassessing Expectations for Blockchain and Development’, Innovations: Technology, Governance, Globalization, vol. 12, no. 1–2, pp. 80–88, Jul. 2018, doi: 10.1162/inov_a_00269.

[24] Q. Zhou, H. Huang, Z. Zheng, and J. Bian, ‘Solutions to Scalability of Blockchain: A Survey’, IEEE Access, vol. 8, pp. 16440–16455, 2020, doi: 10.1109/ACCESS.2020.2967218.

[25] A. Gopalan, A. Sankararaman, A. Walid, and S. Vishwanath, ‘Stability and Scalability of Blockchain Systems’, Proc. ACM Meas. Anal. Comput. Syst., vol. 4, no. 2, p. 35:1-35:35, Jun. 2020, doi: 10.1145/3392153.

[26] D. Khan, L. T. Jung, and M. A. Hashmani, ‘Systematic Literature Review of Challenges in Blockchain Scalability’, Applied Sciences, vol. 11, no. 20, Art. no. 20, Jan. 2021, doi: 10.3390/app11209372.

[27] Y. Xinyi, Z. Yi, and Y. He, ‘Technical Characteristics and Model of Blockchain’, in 2018 10th International Conference on Communication Software and Networks (ICCSN), Jul. 2018, pp. 562–566. doi: 10.1109/ICCSN.2018.8488289.

[28] ‘What is proof of stake (PoS)?’, WhatIs. Accessed: Dec. 30, 2023. [Online]. Available: https://www.techtarget.com/whatis/definition/proof-of-stake-PoS

[29] ‘What is Proof-of-Work (PoW)?’, Ledger. Accessed: Dec. 30, 2023. [Online]. Available: https://www.ledger.com/academy/blockchain/what-is-proof-of-work

[30] S. Amofa *et al.*, ‘A Blockchain-based Architecture Framework for Secure Sharing of Personal Health Data’, in *2018 IEEE 20th International Conference on e-Health Networking, Applications and Services (Healthcom),* Ostrava: IEEE, Sep. 2018, pp. 1–6. doi:10.1109/HealthCom.2018.8531160

[31] L. Huo, D. Jiang, S. Qi, and L. Miao, ‘A Blockchain-Based Security Traffic Measurement Approach to Software Defined Networking’, *Mobile Netw Appl*, vol. 26, no. 2, pp. 586–596, Apr. 2021, doi: 10.1007/s11036-019-01420-6.

[32] D. Mingxiao, M. Xiaofeng, Z. Zhe, W. Xiangwei, and C. Qijun, ‘A review on consensus algorithm of blockchain’, in *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC),* Oct. 2017, pp. 2567–2572. doi: 10.1109/SMC.2017.8123011.

[33] H. D. Zubaydi, Y.-W. Chong, K. Ko, S. M. Hanshi, and S. Karuppayah, ‘A Review on the Role of Blockchain Technology in the Healthcare Domain’, *Electronics*, vol. 8, no. 6, Art. no. 6, Jun. 2019, doi: 10.3390/electronics8060679.

[34] M. M. Merlec, Y. K. Lee, S.-P. Hong, and H. P. In, ‘A Smart Contract-Based Dynamic Consent Management System for Personal Data Usage under GDPR’, *Sensors*, vol. 21, no. 23, p. 7994, Nov. 2021, doi: 10.3390/s21237994.

[35] G.-T. Nguyen and K. Kim, ‘A Survey about Consensus Algorithms Used in Blockchain’, *Journal of Information Processing Systems*, vol. 14, no. 1, pp. 101–128, Feb. 2018, doi: 10.3745/JIPS.01.0024.

[36] S. M. H. Bamakan, A. Motavali, and A. Babaei Bondarti, ‘A survey of blockchain consensus algorithms performance evaluation criteria’, *Expert Systems with Applications*, vol. 154, p. 113385, Sep. 2020, doi: 10.1016/j.eswa.2020.113385.

[37] E. J. De Aguiar, B. S. Faiçal, B. Krishnamachari, and J. Ueyama, ‘A Survey of Blockchain-Based Strategies for Healthcare’, *ACM Comput. Surv.*, vol. 53, no. 2, p. 27:1-27:27, Mar. 2020, doi: 10.1145/3376915.

[38] J. Cheng, L. Xie, X. Tang, N. Xiong, and B. Liu, ‘A survey of security threats and defense on Blockchain’, *Multimed Tools Appl*, vol. 80, no. 20, pp. 30623–30652, Aug. 2021, doi: 10.1007/s11042-020-09368-6.

[39] ‘Blockchain and Kubernetes: A Perfect Match - Collabnix’. Accessed: Jul. 05, 2024. [Online]. Available: https://collabnix.com/blockchain-and-kubernetes-a-perfect-match/

[40] N. Tariq, A. Qamar, M. Asim, and F. A. Khan, ‘Blockchain and Smart Healthcare Security: A Survey’, *Procedia Computer Science*, vol. 175, pp. 615–620, Jan. 2020, doi: 10.1016/j.procs.2020.07.089.

[41] G. Manogaran *et al.*, ‘Blockchain based integrated security measure for reliable service delegation in 6G communication environment’, *Computer Communications*, vol. 161, pp. 248–256, Sep. 2020, doi: 10.1016/j.comcom.2020.07.020.

[42] M. Attaran, ‘Blockchain technology in healthcare: Challenges and opportunities’, *International Journal of Healthcare Management*, vol. 15, no. 1, pp. 70–83, Jan. 2022, doi: 10.1080/20479700.2020.1843887.

[43] I. Roman-Martinez, J. Calvillo-Arbizu, V. J. Mayor-Gallego, G. Madinabeitia-Luque, A. J. Estepa-Alonso, and R. M. Estepa-Alonso, ‘Blockchain-Based Service-Oriented Architecture for Consent Management, Access Control, and Auditing’, *IEEE Access*, vol. 11, pp. 12727–12741, 2023, doi: 10.1109/ACCESS.2023.3242605.

[44] H. H. Jung and F. M. J. Pfister, ‘Blockchain-enabled Clinical Study Consent Management’, *TIM Review*, vol. 10, no. 2, pp. 14–24, Feb. 2020, doi: 10.22215/timreview/1325.

[45] Md. M. H. Onik, S. Aich, J. Yang, C.-S. Kim, and H.-C. Kim, ‘Chapter 8 - Blockchain in Healthcare: Challenges and Solutions’, in *Big Data Analytics for Intelligent Healthcare Management*, N. Dey, H. Das, B. Naik, and H. S. Behera, Eds., in Advances in ubiquitous sensing applications for healthcare. , Academic Press, 2019, pp. 197–226. doi: 10.1016/B978-0-12-818146-1.00008-8.

[46] G. Kaur and C. Gandhi, ‘Chapter 15 - Scalability in Blockchain: Challenges and Solutions’, in *Handbook of Research on Blockchain Technology*, S. Krishnan, V. E. Balas, E. G. Julie, Y. H. Robinson, S. Balaji, and R. Kumar, Eds., Academic Press, 2020, pp. 373–406. doi: 10.1016/B978-0-12-819816-2.00015-0.

[47] L. M. Bach, B. Mihaljevic, and M. Zagar, ‘Comparative analysis of blockchain consensus algorithms’, in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2018, pp. 1545–1550. doi: 10.23919/MIPRO.2018.8400278.

[48] N. Chaudhry and M. M. Yousaf, ‘Consensus Algorithms in Blockchain: Comparative Analysis, Challenges and Opportunities’, in *2018 12th International Conference on Open*

Source Systems and Technologies (ICOSST), Dec. 2018, pp. 54–63. doi: 10.1109/ICOSST.2018.8632190.

[49] N. Aldred, L. Baal, and G. Broda, 'Design and Implementation of a Blockchain-based Consent Management System'.

[50] C. C. Agbo and Q. H. Mahmoud, 'Design and Implementation of a Blockchain-Based E-Health Consent Management Framework', in *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Toronto, ON, Canada: IEEE, Oct. 2020, pp. 812–817. doi: 10.1109/SMC42975.2020.9283203.

[51] S. Srivastava, S. V. Singh, R. B. Singh, and H. K. Shukla, 'Digital Transformation of Healthcare: A Blockchain study', vol. 8, no. 5, 2021.

[52] K. Yaeger, M. Martini, J. Rasouli, and A. Costa, 'Emerging Blockchain Technology Solutions for Modern Healthcare Infrastructure', *Journal of Scientific Innovation in Medicine*, vol. 2, no. 1, p. 1, Jan. 2019, doi: 10.29024/jsim.7.

[53] M. M. Madine *et al.*, 'Fully Decentralized Multi-Party Consent Management for Secure Sharing of Patient Health Records', *IEEE Access*, vol. 8, pp. 225777–225791, 2020, doi: 10.1109/ACCESS.2020.3045048.

[54] M. Al Amin, A. Altarawneh, and I. Ray, 'Informed Consent as Patient Driven Policy for Clinical Diagnosis and Treatment: A Smart Contract Based Approach':, in *Proceedings of the 20th International Conference on Security and Cryptography*, Rome, Italy: SCITEPRESS - Science and Technology Publications, 2023, pp. 159–170. doi: 10.5220/0012090600003555.

[55] J. Moubarak, E. Filiol, and M. Chamoun, 'On blockchain security and relevant attacks', in *2018 IEEE Middle East and North Africa Communications Conference (MENACOMM)*, Apr. 2018, pp. 1–6. doi: 10.1109/MENACOMM.2018.8371010.

[56] G. Karame, 'On the Security and Scalability of Bitcoin's Blockchain', in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 1861–1862. doi: 10.1145/2976749.2976756.

[57] D. Tith *et al.*, 'Patient Consent Management by a Purpose-Based Consent Model for Electronic Health Record Based on Blockchain Technology', *Healthc Inform Res*, vol. 26, no. 4, pp. 265–273, Oct. 2020, doi: 10.4258/hir.2020.26.4.265.

[58] N. A. Asad, Md. T. Elahi, A. A. Hasan, and M. A. Yousuf, 'Permission-Based Blockchain with Proof of Authority for Secured Healthcare Data Sharing', in *2020 2nd International Conference on Advanced Information and Communication Technology (ICAICT)*, Nov. 2020, pp. 35–40. doi: 10.1109/ICAICT51780.2020.9333488.

[59] M. Muzammal, Q. Qu, and B. Nasrulin, 'Renovating blockchain with distributed databases: An open source system', *Future Generation Computer Systems*, vol. 90, pp. 105–117, Jan. 2019, doi: 10.1016/j.future.2018.07.042.

[60] H. Chen, D. Zeng, X. Yan, and C. Xing, Eds., *Smart Health: International Conference, ICSH 2019, Shenzhen, China, July 1–2, 2019, Proceedings*, vol. 11924. in Lecture Notes in Computer Science, vol. 11924. Cham: Springer International Publishing, 2019. doi: 10.1007/978-3-030-34482-5.

[61] S. Khezr, M. Moniruzzaman, A. Yassine, and R. Benlamri, "Blockchain Technology in Healthcare: A Comprehensive Review and Directions for Future Research," *Applied Sciences*, vol. 9, no. 9, p. 1736, Apr. 2019, doi: https://doi.org/10.3390/app9091736.

[62] A. AbuHalimeh and O. Ali, "Comprehensive review for healthcare data quality challenges in blockchain technology," *Frontiers in Big Data*, vol. 6, May 2023, doi: https://doi.org/10.3389/fdata.2023.1173620.

[63] M. Khan, A. Hassan, and Md. I. Ali, 'Secured Insurance Framework Using Blockchain and Smart Contract', *Scientific Programming*, vol. 2021, pp. 1–11, Nov. 2021, doi: 10.1155/2021/6787406.