



Decentralized Content Management System

Adam Gallagher

N00211418

Supervisor: Mohammed Cherbatji

Second Reader: Sue Reardon

Year 4 2024/25

DL836 BSc (Hons) in Creative Computing

Abstract

The focus of this project is the design and implementation of a decentralized content management system (DCMS). By combining decentralized storage and content management systems many of the issues found in a traditional CMS are solved. Examples of these are single points of failure and security vulnerabilities. Additionally, a DCMS would simplify the development of decentralized applications by reducing both complexity and the level of expertise required. The goal of this project was to build a secure DCMS that enables users or organizations to securely manage digital assets in a decentralized way through a user-friendly dashboard and an API integration for other applications. Unlike traditional centralized content management all data is stored using Gun JS a decentralized database, improving security and resilience. Gun JS is a decentralized graph database used to store data across peers in a network. The development of this application required several phases, requirement gathering, design, implementation and testing. Software development best practices were followed during this project including the use of the AGILE framework, splitting the work into individual sprints. For future development could explore improved peer configuring for the storage network and dev ops features such as database rollbacks. The system was successfully developed with key features such as custom digital content creation, API integration and decentralized storage.

Acknowledgements

At this point I would like to take this opportunity to give a massive thank you to my supervisor for this application, Mohammed Cherbati, his support and guidance cannot be undervalued. This project would not be possible without his help. I am also grateful to all my lectures who kept me motivated and helped me throughout this process. I would also like to give a thank you to my fellow students who helped with testing and supporting me during the development of the application.

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Course Director.

WARNING: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not leave copies of your own files on a hard disk where they can be accessed by other. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Creative Computing (Hons) course handbook. Please read carefully and sign the declaration below

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

DECLARATION:

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Student : Adam Gallagher

Signed



Failure to complete and submit this form may lead to an investigation into your work.

Abbreviations and Acronyms

CMS	Content management system
DCMS	Decentralized content management system
DStorage	Decentralized storage
SHA	Secure hash algorithm
AES	Advanced encryption standard
IDB	Indexed database
JWT	JSON web token
UI	User interface
API	Application Programming interface
JSON	JavaScript object notation
TS	Typescript
JS	JavaScript
CSS	Cascading style sheets
HTML	Hypertext markup language

Table of Contents

Chapter 1. Introduction	1
Chapter 2. Research	3
2.1 Introduction	3
2.2 Concept 1 – Decentralized Storage.....	3
2.2.1 Use cases.....	3
2.2.2 Types of Decentralized Storage	4
2.2.3 Views of Decentralized Storage	5
2.3 Concept 2 – Content Management Systems	6
2.3.1 Definition.....	6
2.3.2 Types of CMS.....	6
2.4 Concept 3 – Integrating both concepts together.....	7
2.4.1 Development.....	7
2.4.2 Benefits of a decentralized content management system	8
2.5 Conclusion.....	8
Chapter 3. Requirements	9
3.1 Introduction	9
3.2 Requirements gathering	10
3.2.1 Similar applications	10
3.2.2 Survey.....	11
3.3 Requirements modelling.....	11
3.3.1 Functional requirements.....	11
3.3.2 Non-functional requirements	12
3.4 Feasibility	12
3.5 Conclusion.....	13
Chapter 4. Design.....	14
4.1 Introduction	14
4.2 Program Design.....	14
4.2.1 Technologies	14
4.2.2 Structure of Next JS.....	15
4.2.3 Design Patterns	15
4.2.4 Application architecture (1 page)	16
4.2.5 Database design	17
4.3 User interface design	18
4.3.1 Wireframe	18
4.3.2 User Flow Diagram	19

4.3.3	Style guide.....	20
4.4	Conclusion.....	21
Chapter 5	Implementation.....	22
5.1	Introduction	22
5.2	Scrum Methodology.....	22
5.3	Development environment.....	22
5.4	Demo application	23
5.5	Sprint 1.....	25
5.5.1	Goal	25
5.5.2	Item 1	26
5.5.3	Item 2	26
5.5.4	Item 3	26
5.6	Sprint 2.....	27
5.6.1	Goal	27
5.6.2	Item 1	27
5.6.3	Item 2	28
5.7	Sprint 3.....	31
5.7.1	Goal	31
5.7.2	Item 1	31
5.7.3	Item 2	33
5.7.4	Item 3	34
5.7.5	Item 4	38
5.7.6	Item 5	40
5.7.7	Item 6	41
5.7.8	Item 7	42
5.7.9	Item 8	44
5.8	Sprint 4.....	46
5.8.1	Goal	46
5.8.2	Item 1	46
5.8.3	Item 2	49
5.8.4	Item 3	52
5.8.5	Item 4	53
5.8.6	Item 5	54
5.8.7	Item 6	55
5.9	Sprint 5.....	55
5.9.1	Goal	55

5.9.2	Item 1	56
5.9.3	Item 2	57
5.9.4	Item 3	58
5.10	Sprint 6	60
5.10.1	Goal	60
5.10.2	Item 1	60
5.10.3	Item 2	62
5.11	Sprint 7	63
5.11.1	Goal	63
5.11.2	Item 1	63
5.11.3	Item 2	64
5.11.4	Item 3	66
5.12	Sprint 8	67
5.12.1	Goal	67
5.12.2	Item 1	67
5.12.3	Item 2	68
5.12.4	Item 3	69
5.12.5	Item 4	70
5.12.6	Item 5	71
5.13	Conclusion	72
Chapter 6.	Testing	73
6.1	Introduction	73
6.2	Functional Testing	73
6.2.1	Navigation	73
6.2.2	API	74
6.2.3	UI CRUD	75
6.2.4	Discussion of Functional Testing Results	76
6.3	User Testing	77
6.3.1	Discussion of the user testing	78
6.4	Conclusion	78
Chapter 7.	Project Management	79
7.1	Introduction	79
7.2	Project Phases	79
7.2.1	Proposal	79
7.2.2	Requirements	79
7.2.3	Design	80

7.2.4	Implementation	80
7.2.5	Testing.....	81
7.3	SCRUM Methodology.....	82
7.4	Project Management Tools.....	82
7.4.1	GitHub	82
7.4.2	Journal.....	82
7.5	Reflection	83
7.5.1	Views on the project	83
7.5.2	Working with a supervisor	83
7.5.3	Technical skills.....	83
7.6	Conclusion.....	84
Chapter 8. Conclusion		85
Chapter 9. References.....		86
Chapter 10. Appendix		88

Chapter 1. Introduction

The aim of this project is to develop a decentralized content management system (CMS). Traditional content management systems use centralized systems that are prone to errors and vulnerable to cyber-attacks. These pose serious risks for fields where reliability and accuracy are critical. By using decentralized storage technology this could solve these issues. Storing application data in a fault tolerant decentralized database could help increase efficiency, transparency, and security while also reducing the technical barrier for the creation of decentralized applications. This proposed project would store and manage user's custom digital assets through a user-friendly dashboard and provide an API integration that can be used for external applications for example websites or mobile apps. The research portion of this project aims to explain what decentralized storage and content management systems are with a focus on security and their positive and negative features.

The application is developed using Next JS. This framework combines front end and back-end capabilities so the UI components and API routes can be stored in monolith code base. For data storage Gun JS is used. This is a distributed, peer to peer, decentralized database where data is spread across a network with no centralized authority. The node JS crypto library and the jsonwebtoken (JWT) library are used for handling encryption and decryption in the CMS. These are specifically used for verifying and generating data signatures to protect against unauthorised database tampering. The MetaMask SDK enables connection to the Ethereum network for decentralized authentication in the system. Additionally in the dashboard the Tip Tap library is used for creating a rich text editor component and the IDB library is used to streamline the IndexedDB integration.

To ensure best software development practices and receive continuous feedback the AGILE framework was used. Trello was the primary tool in managing the backlog and tracking progress during the implementation phase of this project. Git/GitHub was used for managing version control in the project. Additionally, a personal journal was kept recording the process. This aided with reflecting on the overall process and problem-solving during development.

Technical and functional requirements were created to aid in the development of this project. The functional requirements focus on the usability of the CMS including features and interactions with the UI. Technical requirements were specific to the development of the application, this includes possible frameworks, databases, authentication and encryption methods. These were selected through a process of researching similar CMS products and testing frameworks and libraries.

The design phase had two primary goals, UI design and the system architecture. For the UI design the Shadcn UI framework was used, providing a colour and typography palettes and a suite of prebuilt components. In Figma through iterative design from wire frames the UI was created. System architecture diagrams were drafted in Figma alongside the UI.

Prior to the start of the implementation phase a demo application was developed to validate the feasibility of a decentralized CMS. This application could read and write custom data to the Gun JS database, serving as a proof of concept. The main implementation phase was split up into two-week sprints following the scrum agile framework. Each sprint had a backlog of tasks with the aim of finishing these by the end of the two weeks. If not complete a task would be pushed into the following sprint. Regular meetings were organised during sprints with the supervisor to track progress and receive feedback. This approach supported continuous feedback and improvements to the overall application.

Testing the application was done using automated jest tests and user testing. The user testing focused on the functionality and usability of the system for developers. Jest validated API routes with mocked responses to ensure proper functionality without a dedicated testing environment. Functional testing was carried out through development process verifying the application performed as expected. To further test the capabilities of the system two sample applications were developed using the CMS as a backend. This demonstrated the viability of the system and find issues with the system.

Chapter 2. Research

2.1 Introduction

This section explores the connection between decentralized storage systems (dStorage) and content management systems (CMS), with a proposal for the development of a decentralized CMS platform. Decentralized CMS offers significant advantages over traditional centralized systems which rely on centralized storage that introduces single points of failure and vulnerability to cyber-attacks. These are seen specifically in relations to cost, data security and system robustness. It achieves this by spreading data across network of nodes.

This paper explores the different types of both decentralized storage and CMS technologies and the views of them in the technology sector. It also investigates how these two technologies could be combined into a single system. This paper highlights potential benefits of a decentralized CMS in industries where data integrity and resilience are vital.

2.2 Concept 1 – Decentralized Storage

2.2.1 Use cases

2.2.1.1 *Internet of things*

One of the most promising uses of decentralized is within the internet of things (IOT) field. A common issue with current IOT technologies is when users are locked into specific vendors specially when they are using different hardware, limiting interoperability and making device integration complex. With decentralized storage users can be connected to any network they require (Chamria, 2024). An example of this is ADEPT (autonomous decentralized peer to peer telemetry) developed by Samsung and IBM this is a decentralized network for IOT for data sharing and coordination between IOT devices.

2.2.1.2 *Public Records*

Decentralized storage shows promise in the relation to storing sensitive government documents for example property records, financial data or tax information. Currently these types of data are stored in siloed centralized database which causes issues with managing this data. Centralized databases are vulnerable to cyber-attacks and manipulation. It is also difficult to gain useful insights into this data. A decentralized approach could enhance security and increase transparency within a government (Chamria, 2024). A real-world example of this can be seen in Brazil with Ubiquity. This is an American based start-up working with the Brazilian Government to update Brazil's real estate records by introducing decentralized storage technology. They aim to make a tamper proof, transparent system for tracking real estate.

2.2.1.3 *Healthcare*

The healthcare industry could benefit from a decentralized system for storing and managing sensitive patient data. Historically, the healthcare industry stored all its data in paper records but in recent years it has switched to digital records. Despite switching to a digitized system there remains issues. Traditional centralized digital storage is prone to unauthorized users and data breaches. A decentralized approach could offer stronger security, prevent tampering of records and build trust among health care workers and patients. This in turn could potentially reduce mistakes and provide a

better service (Chamria, 2024). An example of this can be seen with an American based blockchain start-up called Gem. Gem uses blockchain technology to provide real-time, secure access to medical records. Using Ethereum smart contracts Gem provides patients with control over their own health records while allowing for secure sharing of data with authorized professionals. This promotes a patient first approach to healthcare record management.

2.2.2 Types of Decentralized Storage

2.2.2.1 Blockchain storage

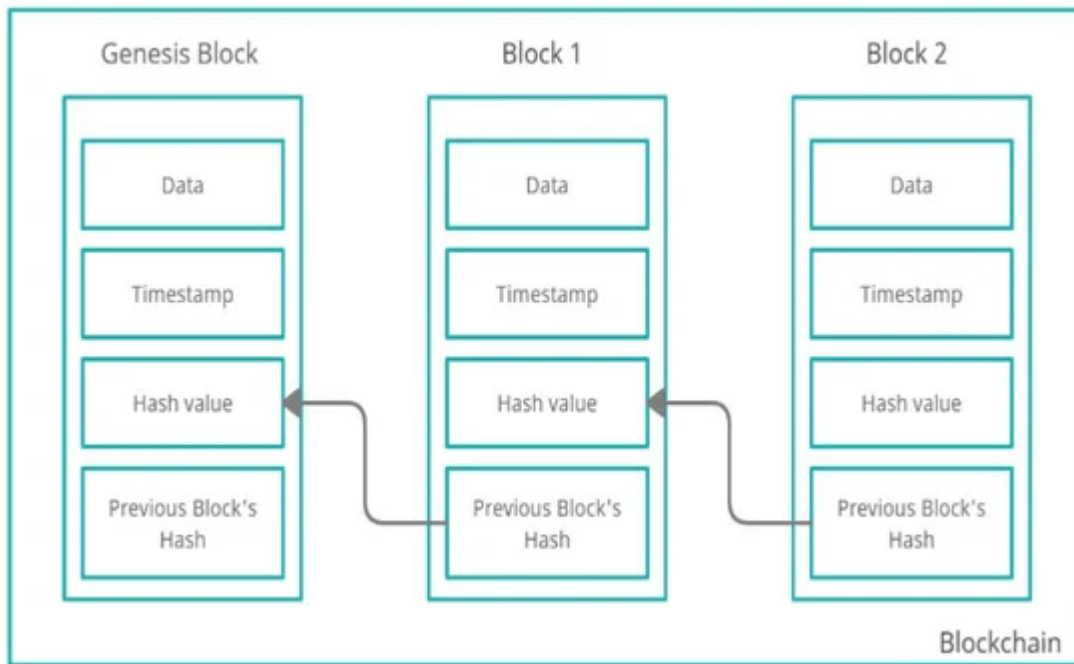


Figure 22-1 Blockchain diagram.

The most widely recognized solution for decentralized storage is blockchain technology. Blockchain uses distributed ledger technology (DLT). The DLT serves as a decentralized database of transactions between different parties in chronological order. Each operation is grouped into a block, which is cryptographically linked by a pointer to the previous block creating a chain of blocks (See Figure 2-1). This data structure ensures traceability. Additionally, Sharding is used in this technology for optimisation. This is the process of dividing files into smaller parts (shards) then each shard is duplicated across multiple nodes in the network to prevent data loss during transmission. Additionally, the data is encrypted using private keys preventing the sensitive data being viewed by other nodes in the network. Every interaction is permanently recorded in the ledger; this enables the system to confirm and synchronise transaction data across the nodes in the blockchain. The data in blockchain is immutable and designed to save these interactions forever (Moore, 2023).

Despite these advantages there is a significant flaw with blockchain technology in relation to scaling and storing large files. It is estimated that storing one megabyte of data can cost up to seventeen thousand US dollars due to the cost of energy (Pinto, 2020).

2.2.2.2 Decentralized cloud storage

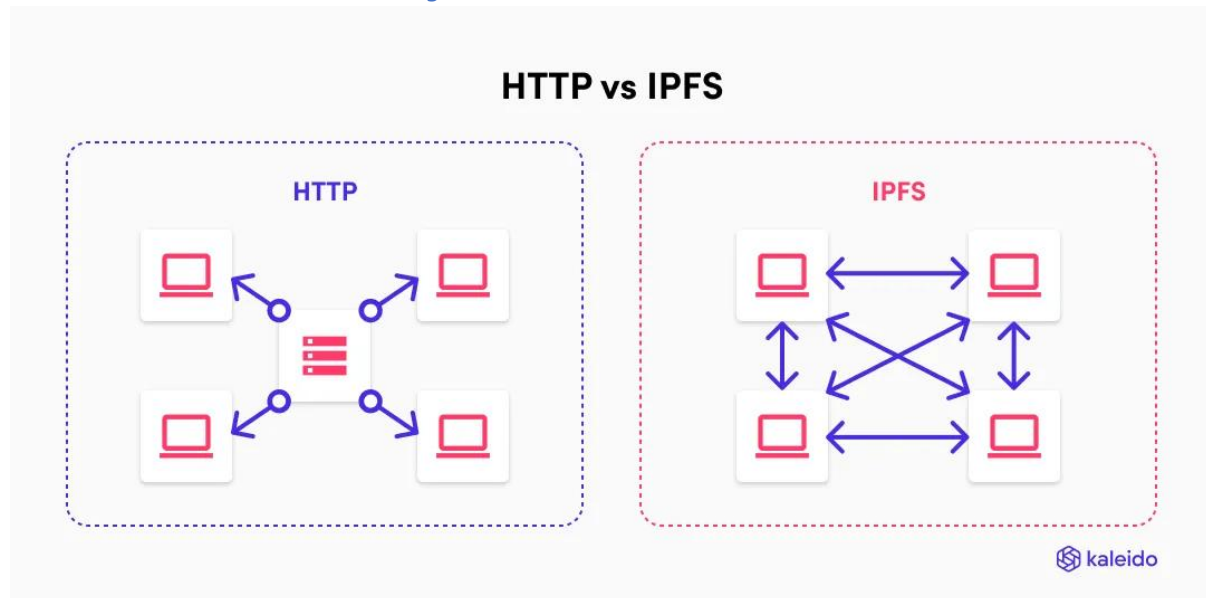


Figure 2-2 IPFS diagram

Another decentralized storage solution is decentralized cloud storage which works similarly to blockchain by distributing data across a network of nodes (see Figure 2-2). Unlike traditional centralized servers' data can be retrieved from a nearby node reducing latency and possibly improving speed. These systems often use the Interplanetary File Storage Protocol (IPFS). IPFS is a peer-to-peer protocol designed to replace conventional hypermedia protocols such as HTTP and HTTPS. IPFS enables users to access and share large volumes of data through content addressing as opposed to location-based addressing. When a file is added to IPFS a hash is created, this is a unique identifier for the data. This type of cryptographic data is a Merkle Directed Acyclic Graph (DAG). This is a special data structure for modelling other types of data structures. This is optimised to remove duplicate content in the network. The purpose of IPFS is to connect all nodes in a network with a centralized approach to share all data.

Additionally, for easing the access to data IPFS uses the Inter Planetary Naming System (IPNS). This is the process of connecting the hash code for content in the network with a readable format. When data is requested from the network, the network calculates which node holds the required content and retrieves it (*IPFS Documentation / IPFS Docs*, n.d.).

2.2.3 Views of Decentralized Storage

2.2.3.1 Props and cons of decentralized systems

One of the primary benefits of a decentralized system is its high reliability. When data is spread among multiple hosts in a network, redundant copies are stored as well. This eliminates a single point of failure. So, if there is a hardware failure, backup copies of the data are available. Another benefit is its reduced cost. In a decentralized network, machine requirements are lower, which then reduces the need for expensive high-performance hardware and software. There is also the possibility for millions of peers to store data in a decentralized network, significantly increasing the availability of storage space.

However, there are some issues with this approach. A major concern is the lack of trust with peer-to-peer systems, this exists for several reasons. By distributing data, it bypasses centralized regulations. Additionally, there is a lack of accountability in the occurrence of data loss, where a centralized solution has clear accountability (A Comprehensive Survey on Blockchain-Based Decentralized Storage Networks, 2023).

2.3 Concept 2 – Content Management Systems

2.3.1 Definition

2.3.1.1 *What is a CMS?*

A content management system (CMS) is a piece of software that helps users manage digital content in a simple efficient way. Teams can use a CMS to create, edit, organize and publish their content. It acts as a single point of contact to store data and provide automated processes for collaborative content management and creation. Some examples of popular CMS are Pocketbase, Joomla and Drupal. (Who, what, and types of content management systems? n.d.).

2.3.1.2 *Key features of a CMS*

One of the primary features of a CMS is the ability to assign different user roles. This functionality enables users to access appropriate digital content relevant to their responsibilities. These roles can include standard organization roles, application management roles and resource-based roles that define permissions. The main function of a CMS is digital asset management serving as a central hub for digital assets to be managed and to create rules and workflows to define how the content can be utilized. Every asset created must enforce a strict type for that asset's custom attributes.

2.3.2 Types of CMS

2.3.2.1 *Headless*

A headless CMS separates the content management backend from the front-end presentation layer. This architecture provides the user with the freedom and flexibility to create and manage content with their own external applications or other specific use cases. One of the advantages of a headless CMS is that users can manage their content centrally and distribute it across multiple channels such as websites or mobile applications (Osman, 2024).

2.3.2.2 *WCMS*

Another common type of CMS is the web content management system (WCMS). A WCMS allows users to manage digital components of a website without requiring any knowledge of web development technologies such as HTML or JavaScript. Unlike other CMS which deal with content across multiple channels such as web or print, a WCMS can only manage web content. (Jones, 2024).

2.3.2.3 *ECMS*

Enterprise content management systems (ECMS) are also commonly used. These systems collect and organize an organization's documentation. They ensure important information is delivered to the correct audiences within the organization such as employees, customers or business stakeholders.

An ECMS enables all members of a company to access the content they need to complete projects and make important business decisions. The primary benefit and purpose of this type of software is to increase efficiency and productivity by providing easy access to digital content. (Jones, 2024).

2.4 Concept 3 – Integrating both concepts together

2.4.1 Development

2.4.1.1 User authentication

For user authentication MetaMask will be integrated into the system. MetaMask is a cryptocurrency wallet that provides a browser extension designed for interacting with the Ethereum blockchain. By using this extension, traditional centralized authentication methods such as usernames and passwords are not needed. Instead, the user authenticates by connecting to their wallets and retrieving their cryptographic signature (Use MetaMask SDK with React UI | MetaMask Developer Documentation, n.d.). This approach benefits user privacy and security because user data is not stored in a centralized server which reduces risk of data breaches.

2.4.1.2 Data storage

For storing the users generated data Gun JS will be used. Gun JS is a peer-to-peer, decentralized database designed for synchronisation in distributed systems. It operates based on the principles similar blockchain and decentralized cloud storage defined in the previous section (See Chapter 2.2.2). Gun JS is known for its peer-to-peer data synchronisation which enables multiple nodes in a network to update data. This includes peers that temporarily loses connection, it does this by caching data locally on the peer's device and automatically syncing it when connection is restored. Additionally, this database uses a graph model which allows for flexible storage of complex data. Gun JS has a built-in cryptographic library named Security, Encryption, Authorization (SEA). This ensures privacy and provides user authentication. (GUN — the Database for Freedom Fighters - Docs v2.0, n.d.).

2.4.1.3 Dashboard / API development

For the dashboard UI and API development the Next JS framework will be used. Some of the key features of Next JS includes support for server side and client-side rendering of components which enhance the performance of application allowing many of the API calls to be made on the server and delivered to the client. Next JS offers both UI and API capabilities which makes it well suited for making a headless CMS dashboard. It also supports custom middleware and Typescript both of which improve the development process. It's important to note that the entry point to the data, dashboard and API being developed can run locally or be hosted as a centralized application, however the user's generated data is stored in a decentralized style (Next.Js by Vercel - the React Framework, n.d.).

2.4.1.4 Styling

Next JS, by default is configured to use Tailwind CSS as a styling framework. Tailwind offers utility classes that can be used for style components efficiently without writing vanilla CSS. This will be used in conjunction the Shadcn component library for UI design. Shadcn was chosen as it has many

pre-built components and its compatible with React and Tailwind CSS. This will greatly improve the development process. (Shadcn, n.d.).

2.4.2 Benefits of a decentralized content management system

The primary and most important benefits of a decentralized content management system (DCMS) are seen in its increased security. By distributing the data among peers in a network it makes it harder for malicious users to compromise the system. If one or more of the peers, go offline the content will still be accessible via the other peers in the network. These factors contribute to increased privacy of users. Another benefit of this is the reduced cost as the user is not relying on a centralized server for storing their data. (Gagan, 2023). Additionally, a DCMS would streamline the process of building decentralized applications by reducing the need for specialized technical knowledge.

2.5 Conclusion

Through examining the current state of decentralized storage and CMS technology, the research section of this paper has outlined the potential for a decentralized CMS (DCMS) and the inherent benefits that come with the merging of these two technologies. A DCMS addresses the vulnerabilities of traditional centralized content management such as single points of failure, lower reliability and lack of transparency. Decentralized technology resolves many of these issues while reducing the cost of running a CMS. As discussed above, decentralized storage is not without flaws due to its circumvention of certain regulations and lack of accountability for data loss or outages. Despite these challenges, developers are starting to recognize the flaws with traditional centralized systems. The proposal of a decentralized content management system highlights the promising future of greater adoption of decentralized systems especially in areas where access and cost are important like healthcare, public records and internet of things.

Chapter 3. Requirements

3.1 Introduction

The requirements phase is necessary to the development process, as it informs what the application should be capable of from a usability and technical perspective. It ensures that the system is created with user needs as opposed to developer assumptions. The aim of this project is to develop a decentralized CMS with a focus on security.

The primary functional requirement for the decentralized CMS is to allow users to manage custom digital assets with their own custom properties and values. Every digital asset will have full CRUD capabilities (Create, Read, Update, Delete). As a headless CMS the platform provides an automatically generated API integration which can be used in external projects. The dashboard and API will have collections with properties and single valued resource routes. With this custom, dynamic API documentation will be generated on the platform to ensure a quality developer experience. There will also be a login and registration flow integrated into the API for decentralized user authentication. API logs will be saved to the dashboard to ease with bug fixing in external applications making API requests.

Separate from the API registration the dashboard user authentication will be handled using the MetaMask SDK which connects to the Ethereum network for blockchain based decentralized user authentication.

Security and decentralized storage are the main goals for the technical requirements. User created data will be stored in Gun JS a distributed, peer to peer, decentralized database where data is spread across a network with no centralized authority. To ensure security all user generated data is encrypted using a combination of AES (Advanced Encryption Standard) and RSA for handling public and private keys. Additionally, every individual entry to the database is signed with a unique digital signature to verify the authenticity of the data.

3.2 Requirements gathering

3.2.1 Similar applications

3.2.1.1 Pocket base

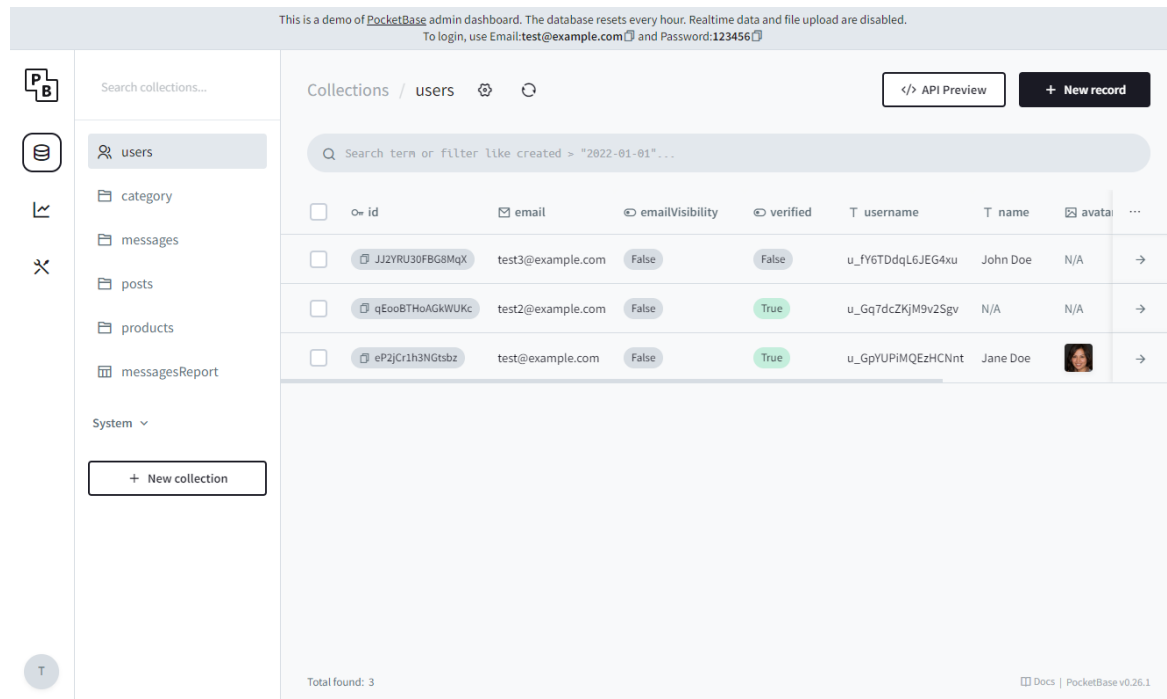


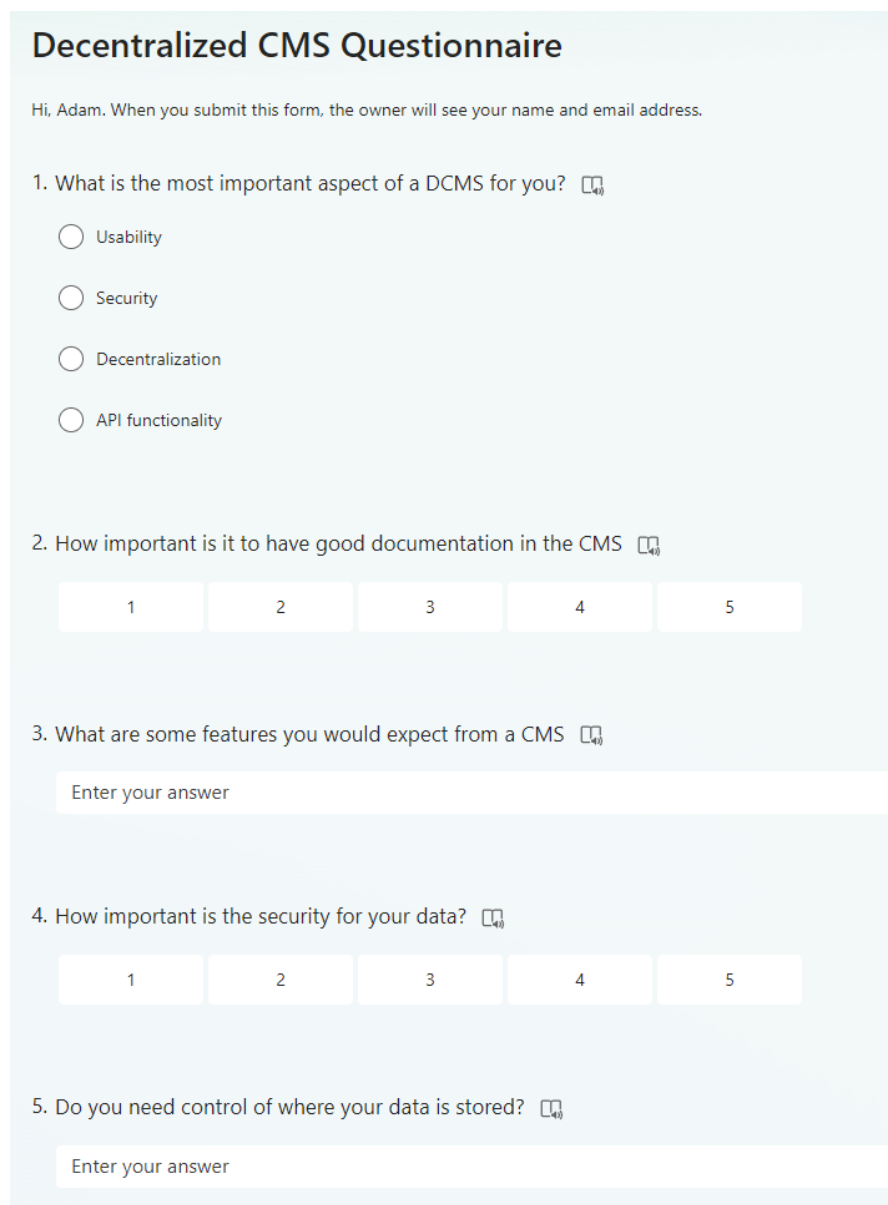
Figure 3-1 Pocketbase CMS.

One of the first CMS applications that was studied was Pocketbase. Pocketbase is an open-source lightweight headless CMS. Pocketbase provides core backend functionalities like authentication, data storage and file management making it well suited for smaller low complexity projects. The main benefits of this CMS are its simplicity and ease of deployment. All the application data is stored in one SQLite file which is a lightweight serverless relational database. This is what enables it to be deployed without a complex backend.

Another major strength of this Pocketbase is its comprehensive documentation. The documentation provides usable detailed guides and practical code snippets covering many different situations and use cases.


While Pocketbase is easy to use and effective for smaller applications its centralized approach and SQLite architecture limit its resilience. Despite this Pocketbase simplicity and user-friendly design gives great insight into the functional requirements for a CMS (See figure 3-1).

3.2.2 Survey




Decentralized CMS Questionnaire

Hi, Adam. When you submit this form, the owner will see your name and email address.

1. What is the most important aspect of a DCMS for you? 

- ☐ Usability
- ☐ Security
- ☐ Decentralization
- ☐ API functionality

2. How important is it to have good documentation in the CMS 


1

2


3

4

5

3. What are some features you would expect from a CMS 

Enter your answer

4. How important is the security for your data? 


1

2

3

4

5

5. Do you need control of where your data is stored? 

Enter your answer

Figure 3-2 CMS questionnaire.

To support the requirement gathering a survey was created for the application. It was distributed to several software developers – the intended users for the application (See figure 3-2). The goal of this was to gain insights into the required features and find out what aspects are most important to the users. Some of the core features found from this includes clear dynamic documentation based on created resources, built in routes for user authentication in the API, a devops page for seeing the status of API requests and the ability to configure a decentralized network.

3.3 Requirements modelling

3.3.1 Functional requirements

1. Store Data in a decentralized database with encryption decryption functionalities. This includes integrity verification using data signatures and generating API keys.

2. Define custom collections with unique properties and values with full CRUD functionality.
3. Create single variable data resource with full CRUD functionality.
4. The developer using the CMS should be able to authenticate themselves using the MetaMask wallet SDK, there should also be a separate login / registration flow for users authenticating from external applications that are using the CMS as a backend.
5. The application should have an API integration which can be used outside of the dashboard in external applications for a backend.
6. The user should be able to see all rows in dashboard and have custom documentation with its end points automatically generated.
7. API calls should be stored in the application so the developer can see when and what type of calls were made. This will aid in bug fixing for developers using the CMS as a backend.
8. The CMS will use the provided Gun JS public peer by default, but the dashboard should have the functionality to configure multiple other peers in the storage network.

3.3.2 Non-functional requirements

Nonfunctional requirements are vital for ensuring a quality application. This is done by prioritising quality, efficacy and reliability. These do not directly affect the core functionality of the CMS but are still necessary. In this section the focus will primarily be on security and performance.

Security is a major concern in this application especially due to the decentralized nature of Gun JS. In these types of systems, no single peer has supremacy over the data stored. Each peer in the network has equal rights to alter or create data. With this comes a risk of unauthorised manipulation of the data. To prevent this the application will use a combination of AES and RSA encryption the Node.js crypto library ensuring data is not stored as readable plain text. Decryption is only possible with the correct RSA private keys.

Additionally, to ensure data integrity, every entry added into the database has a unique data signature. The signature is generated by stringifying the data then encrypting it. When data is returned from the database, both the response and the data signature are decrypted and compared. A mismatch could signify data manipulation from an unauthorized user. With this then a rollback could be performed.

Next JS provides preloading of components on the server before serving the client this can increase the performance as the browser does not have to fetch and load all the client components improving page loading times and enhancing the overall user experience.

3.4 Feasibility

Next JS 15.2.3 is a React framework designed for building full-stack web applications. It utilizes React Components to build user interfaces, and Next JS provides additional features and optimisations. Next JS supports built in API routing and optimisations on the React 19 library, such as the inclusion of server-side rendering of UI. With both front end and back-end capabilities this will be the foundation for the CMS.

For data storage Gun JS will be used. Gun JS is a distributed decentralized database that operates by caching data on multiple peers in a network and synchronizing when data updates occur. By distributing the data across multiple peers with no single peer with overall control the data storage becomes decentralized. Due to the decentralized nature of the system, encryption and authentication is critical. Gun JS includes a security encryption and authentication (SEA) framework, but it will not be used in this project due to compatibility issues with Next JS. To address this compatibility, issue the node JS crypto library will be utilized for generating public and private RSA key pairs for encrypting and decrypting private data. For secure authentication on the server the Json web token (JWT) library will be utilized. To further ensure security the application will use a combination of AES for encrypting content and RSA for secure key pairs. Additionally, every entry will be signed with a unique digital signature ensuring the system can verify data has not been tampered with.

For authenticating users on the client providing access to the dashboard the MetaMask SDK will be utilized. This SDK is specially designed for React and enables connection to the MetaMask wallet and Ethereum network for decentralised authentication. However, there is a compatibility issue with React version 19. When using Node version 23.8.0 and NPM version 10.9.0 the SDK will cause an invalid dependency tree during package installation. This error does not prevent the application from working. To resolve this the --force flag must be added when installing Node packages inside the application.

For testing the API routes the Jest testing framework will be used. Jest has the function to mock responses and simulate behaviour with API routes, including logging without mutation of the actual database. This enables safe and isolated testing eliminating the need for a dedicated testing environment or database.

Design is not the primary focus of this project, therefore for simplicity and convenience, the Shadcn component library was selected. Shadcn provides pre-styled components, many of which come with built in functionality such as the data tables filtering and sorting methods. Shadcn is built using Tailwind CSS, so the components are easily customizable for the specific use cases within the application.

3.5 Conclusion

The requirements section of this paper has clearly displayed the technical and functional requirements for developing the proposed decentralized content management system. Functional requirements such as the creation of custom user defined digital assets and dynamic API generation have been identified as core features to the systems usability. The technical requirements focus on security and the need for AES and RSA encryption, digital signatures and decentralized storage using Gun JS. User authentication is handled through the integration of the MetaMask SDK. This offers block chain-based user verification. Auto-generating API routes and documentation based on user generated data is needed for the usability of this system. These requirements are the foundations for this system and inform the following phases in this project.

Chapter 4. Design

4.1 Introduction

This chapter outlines the design for the decentralized content management system (DCMS). The goal of this chapter is to provide a clear plan to ensure the application meets the functional and technical requirements established in the previous chapter (See chapter 3).

There are two primary goals in this phase. The first is to design the user interface (UI) based on the requirements. The second goal is to develop the system architecture for the system. The UI was created using the Figma design tool through an iterative process, ensuring the required features were incorporated. The Shadcn UI library was utilized to support this phase by offering pre-built components with a typography and colour palettes.

Concurrently with the UI design, the system architecture was designed with a similar iterative approach. The goal of the design was to plan for the implementation phase and design a system that matched the technical requirements.

4.2 Program Design

This section outlines the technical design of the project, setting the foundations for the development of the decentralized content management system. The primary goal for this was to prepare for the development phase. The technologies chosen during the requirements Next JS, Gun JS and MetaMask informed the systems architecture. This was crucial in verifying the feasibility of the application by establishing plan for the development.

4.2.1 Technologies

The primary technology in this project is the Gun JS database, a graph-based peer-to-peer, decentralized database designed for synchronisation in distributed systems. This database works by caching data on across multiple nodes or peers in a network. Other decentralized databases were considered, such as Orbit DB and the IPFS protocol. Ultimately, Gun JS was chosen for its usability and suite of built in libraries such as SEA (Security, Encryption, Authentication). Despite the SEA libraries advantages this library was not actually utilized in this project due to compatibility issues.

To solve the encryption and decryption requirements two Node JS libraries were used, crypto and JWT. These provide the generation of API keys using RSA public and private keys, ensuring secure authenticated access to the API. The purpose of these was to secure data in the decentralized database as any peer could potentially modify data. To further prevent unauthorized data mutation the crypto library was used to enforce data signatures, ensuring the integrity of data. Additionally, AES encryption will be used alongside RSA encryption. This was chosen as it has no size limit for encryption like RSA encryption.

The MetaMask wallet SDK is used for authenticating the user in a decentralized way. This SDK was chosen as it offers a developer-friendly method for authenticating in decentralized applications. This helps streamline the development process and giving the application access to the public wallet address of the user which is needed for performing encryption and decryption.

The project will be built in the Next JS framework. Next JS provides front-end and back-end capabilities allowing the dashboard and API to co-exist in the same codebase. This prevents issues

with hosting multiple codebases for one project, creating a better developer experience. Typescript was chosen which offers the flexibility of vanilla JavaScript but with a strict type system that helps prevent logical errors in the code.

4.2.2 Structure of Next JS

The bulk of the Next JS application is contained within the `src` directory. Within this directory there are several sub directories, the `types` directory holds the Typescript interfaces for enforcing a strongly typed code base. The `Utils` directory which contains three files `api.ts`, `index.ts` and `security.ts`. `Index.ts` is for generic utility functions which are used throughout the application. `Api.ts` and `security.ts` are for utility functions that are specific the API integration and the security aspect of the application. Additionally, the `src` directory contains the `UI components` directory which holds all the custom components and installed Shadcn prebuilt components.

In this `src` directory, there is an `app` folder this specifically holds the application logic such as the page routing and API. Next JS uses directory-based routing and can handle both front-end and back-end capabilities. There is an `API` directory within the `app` folder, and each subdirectory in this represents an API route. Each one of these routes will have a `route.ts` file which defines what code will run depending on the type of http request made to it (e.g. POST, GET, PUT, DELETE).

There is a `collection` directory for handling CRUD operations on user-defined collections. Similarly, there is a `singles` directory for handling CRUD operations on user defined single-valued database entries. There is also two separate authentication folders named `auth` and `userAuth`. The `Auth` route is for handling the first-time authentication of the developer using the dashboard. It stores the users MetaMask wallet address and generates an `ENV` file with the default parameters needed to run the CMS. The `userAuth` route is used to handle registration and login by real users accessing the backend through an external application using the CMS as a back-end. This route manages unique registration and returns a JWT token which can be validated.

In the root of the project outside the `src` directory various config files are used to configure the application, for example, `package.json`, `.env` and `tailwind.config.ts`. Additionally, in the root directory there is a `public` folder. The `public` folder is used to store app data including configured peers and the users MetaMask wallet address. Additionally, for storing user defined models (representation of properties for a row in DB) for database entries are stored in the `IndexedDB` on the client using the `idb` library. No actual database entries are stored in the `public` directory or the `IndexedDB`.

There is another directory named `radata`, which is where the Gun JS entries are cached from the server. When Gun JS data is accessed from the client, the data is cached in the browser's local storage. Within `radata`, there is a JSON file named `!.json`, which holds the Gun JS data is a graph representation. It's worth noting when Gun JS is run in dev mode many temporary `radata` files are written into the root of the project. This happens when there is a synchronization between the peers in the network. The temporary files only show up in development mode due to write access restrictions. When the application is built this behaviour does not occur.

4.2.3 Design Patterns

The project structure follows Next JS best practices, utilizing the monolith design pattern. In this pattern, all elements of the application- front-end, back-end and testing are bundled together into a single codebase. The application also uses file-based routing to handle both the API routes and the

page routing. This simplifies the design of the project and improves the developer experience removing the need to host multiple projects for one CMS instance.

4.2.4 Application architecture (1 page)

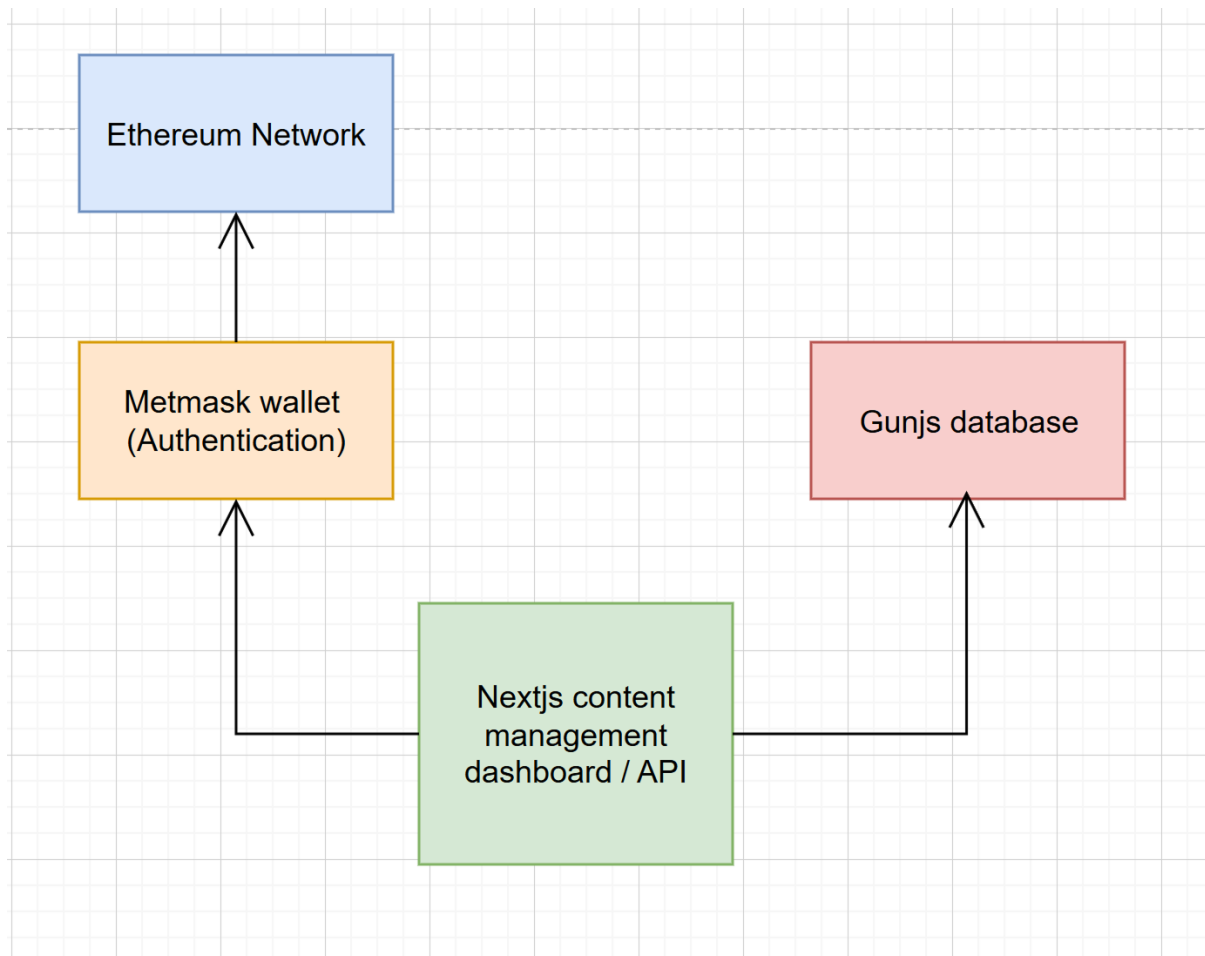


Figure 4-1 Application architecture.

The basic architecture overview of the application consists of a Next JS dashboard, where users can authenticate using MetaMask wallet authentication. From this dashboard, the user can view their data, which is stored in a Gun JS database. This works by distributing data across a network of peers. Thanks to Next JS file-based routing, the hosted dashboard also functions as an API integration allowing external applications to interact with it (See figure 4-1).

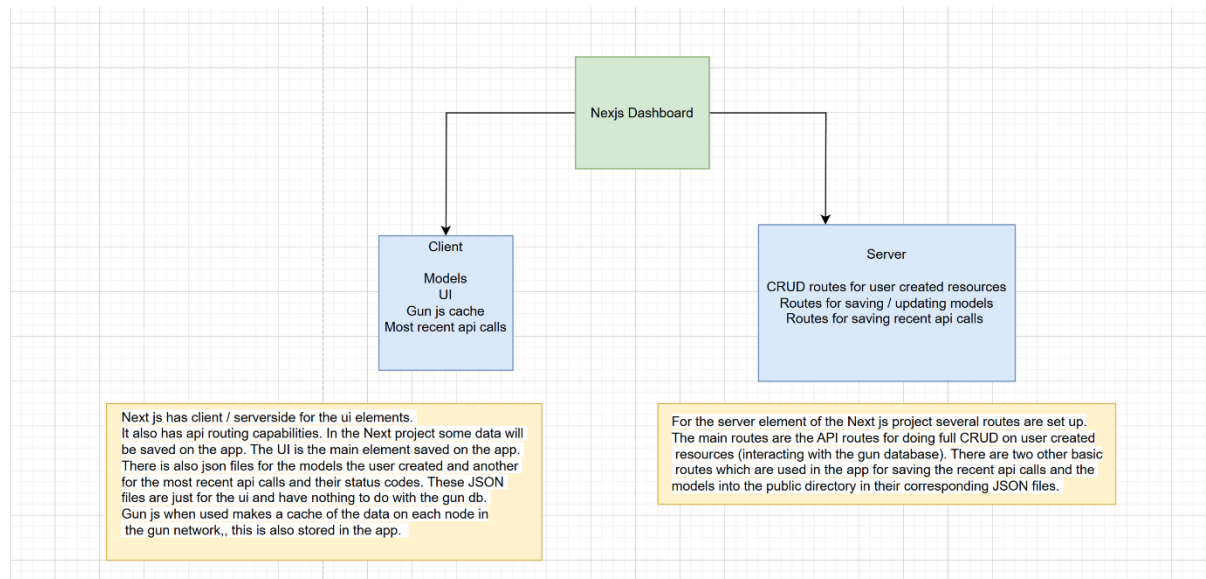


Figure 4-2 Client server split for the CMS.

The application is built as a monolithic Next JS project with both the front-end and back-end all contained under the same codebase. The server portion handles all the CRUD routes for collections, singles, and external application user authentication. The server also includes helper routes to provide the front end with additional functionalities. Examples include saving API response statuses and generating the .env file during initial setup.

The client side is responsible for rendering the UI components. From the client users can view all stored data and interact with the endpoints via the dashboard UI. Additionally, a documentation page is automatically generated for the user defined collections and singles, improving the experience of using the CMS (See figure 4-2).

4.2.5 Database design

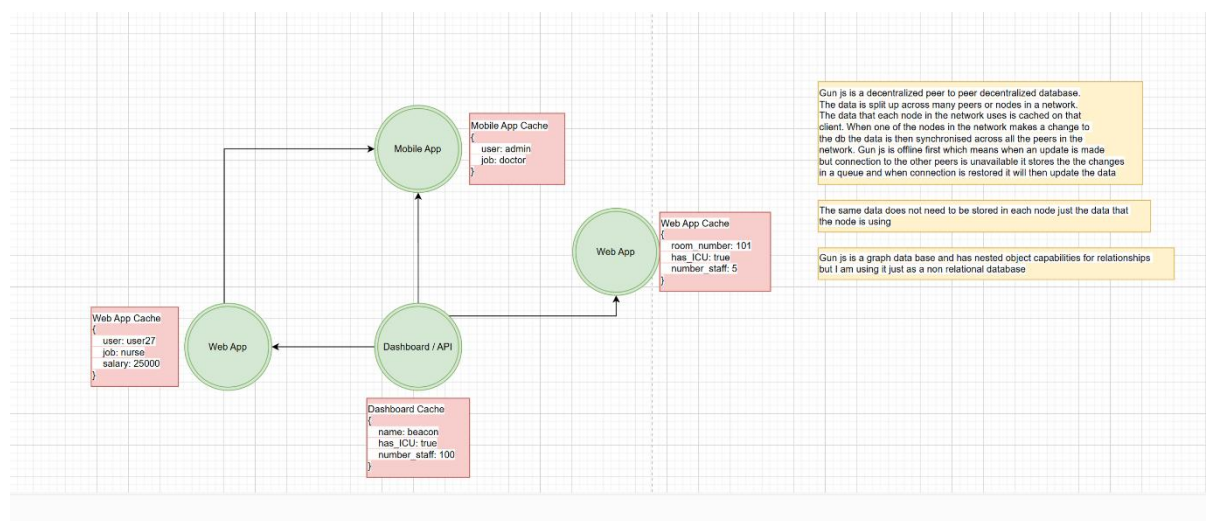


Figure 4-3 Diagram for how Gun JS network works.

Gun JS is a decentralized database made in JavaScript that stores data in the cache of nodes across a network. None of these nodes have centralized authority over the data. Meaning any peer in the

network can update data. When data is updated, Gun JS synchronises data across the other peers in the network. Gun JS is offline-first, meaning if a peer if a peer loses connection, changes stay in the local cache then are synced with the network once the connection is restored. Gun JS is a graph database, where all entries are connected through relationships. However, the graph capabilities are not utilized in the CMS. The data base is used in the same way as a non-relational database just for decentralized storage (See figure 4-3).

4.3 User interface design

This section focuses on the process of designing the user interface. The Figma design tool was used for this section. The designs created using the requirements and research sections to ensure a useable implementation that meets the required functionality. The design process followed an iterative approach starting with basic wireframes (See figure 4-8). With every iteration the design was refined further.

4.3.1 Wireframe

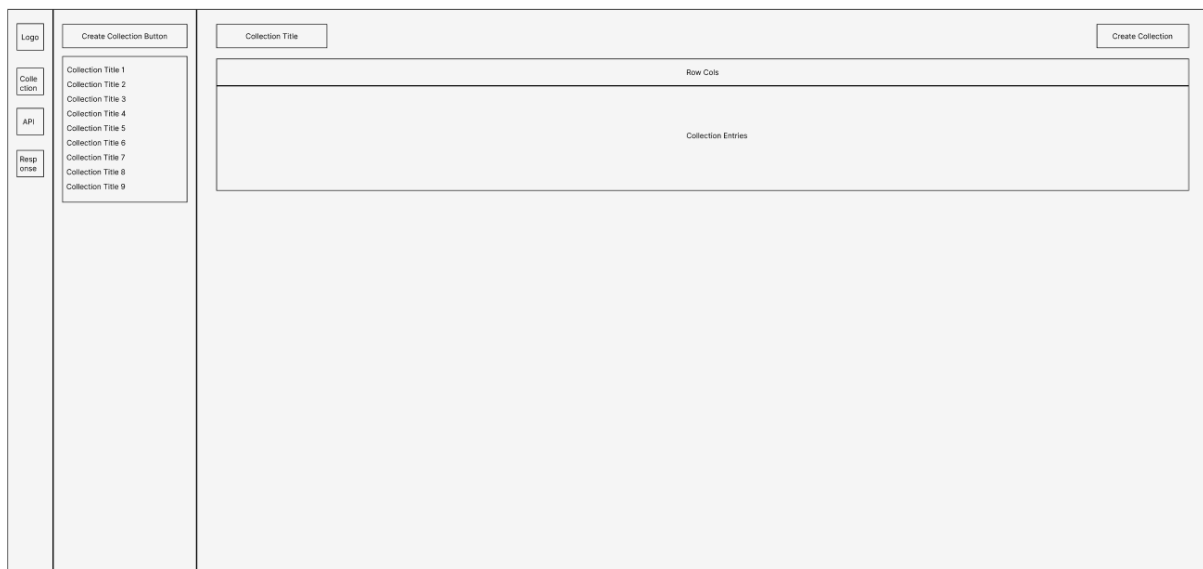


Figure 4-4 Database page wireframe

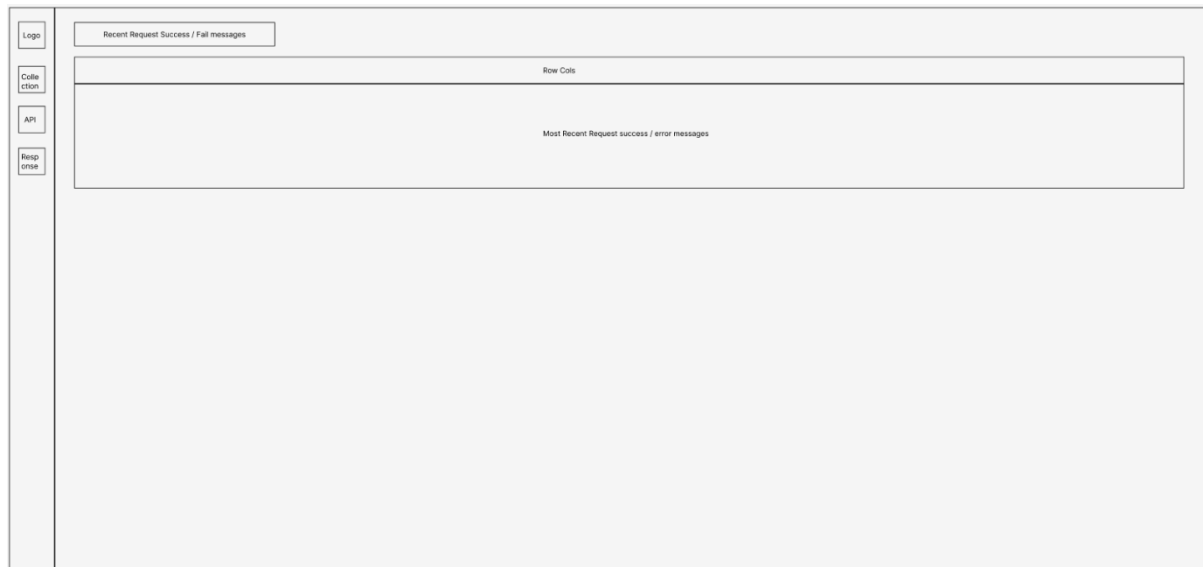


Figure 4-5 Status page wireframe.

The first stage in the UI design process was wireframing. A wireframe is a barebones representation of the UI that does not contain colour or typography. It is used to rapidly prototype the core features of the application. Above you can see the side navigation bar which is used for switching between different pages in the CMS (See figure 4-4). Additionally, the collection inner side bar can be seen. This inner bar which is used for navigation between the user created collections along with a button to add a new row. All user generated rows will be shown with their defined properties in a table component. This wire frame served as a first draft, highlighting primary features that needed to be included such as the create collection component form and the controls for doing full CRUD operations.

4.3.2 User Flow Diagram

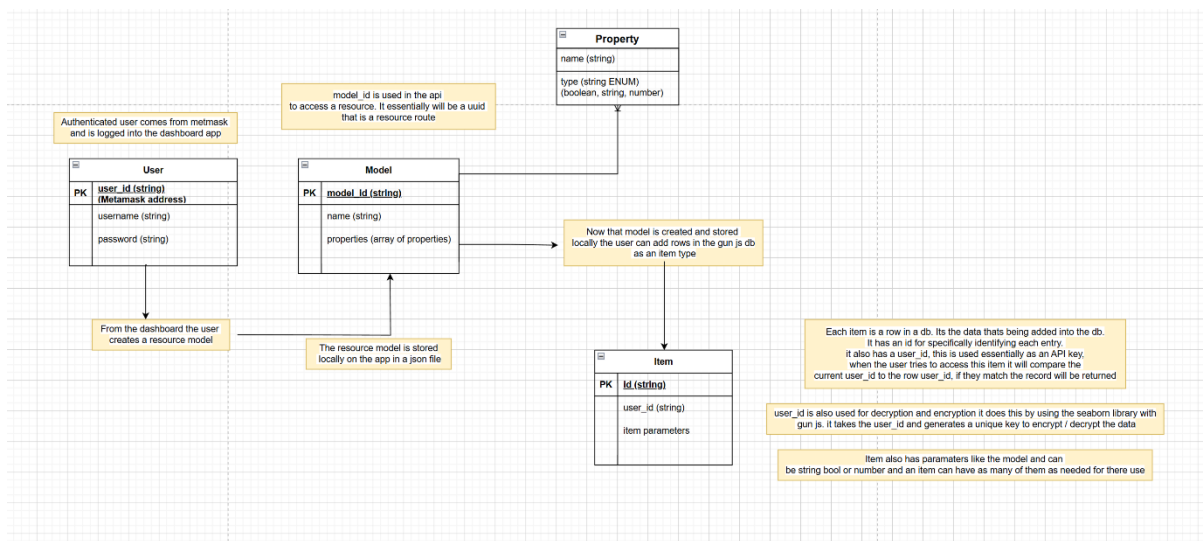


Figure 4-6 User flow diagram.

This user flow diagram was created to provide a clearer understanding of how data is saved and used within the application (See figure 4-6). Starting from the left when a user is authenticated, the

user object is stored. This object contains information about the user. Once authenticated, a user can then define models, which consists of custom properties. Each property consists of a name and a type, which can be either string, number, Boolean or rich text. This is done to enforce strict type validation. Items are added to the database with values for their defined properties.

4.3.3 Style guide

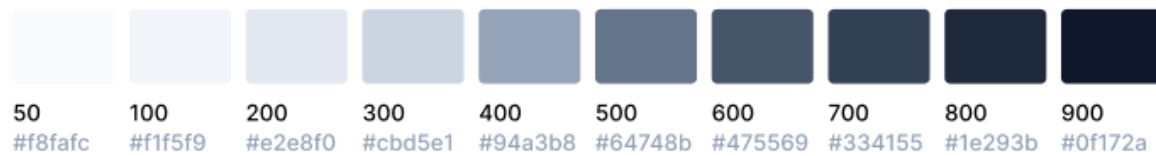


Figure 4-7 Colour palette.

h1

Taxing Laughter: The Joke Tax Chronicles

h2

The People of the Kingdom

h3

The Joke Tax

h4

People stopped telling jokes

p

The king, seeing how much happier his subjects were, realized the error of his ways and repealed the joke tax.

list

- 1st level of puns: 5 gold coins
- 2nd level of jokes: 10 gold coins
- 3rd level of one-liners : 20 gold coins

lead

A modal dialog that interrupts the user with important content and expects a response.

small

Email address

subtle

Enter your email address.

Figure 4-8 Typography palette.

Shadcn provides a typography and colour scheme (see figures 4-7, 4-8), which were used through the application design to simplify the design and development processes. Since Shadcn is built with Tailwind CSS, the application also uses Tailwind spacing variables to provide a consistent spacing system across the components and pages.

4.4 Conclusion

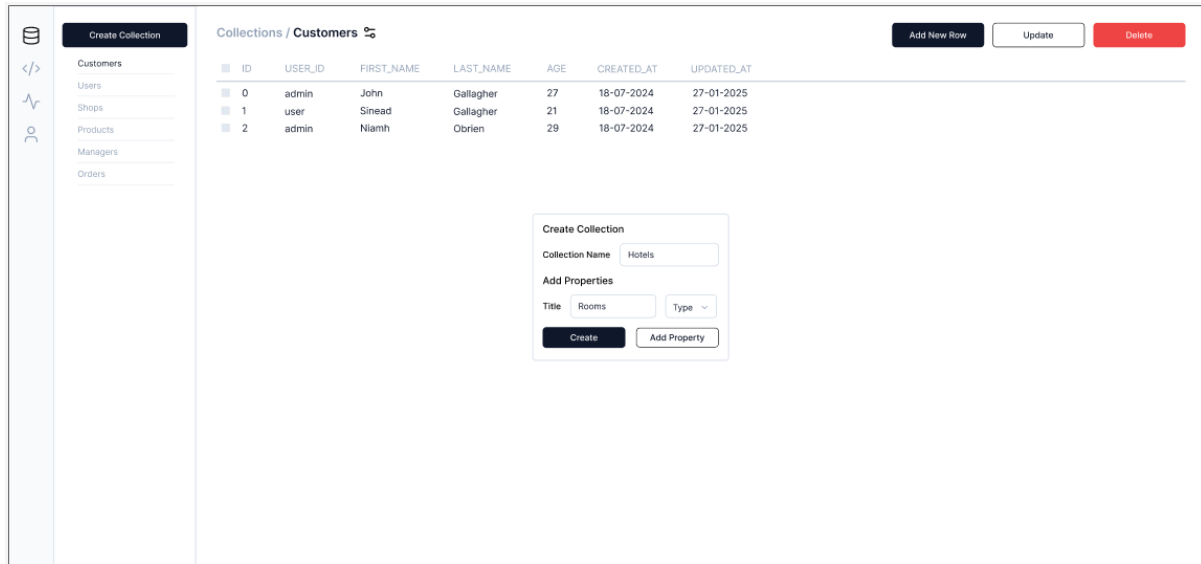


Figure 4-9 Final iteration database page

A final design for the UI portion was created through the iterative process, evolving from wire frames to the final high-fidelity prototype. The final prototype incorporates the functional requirements gathered from the research and requirements phases of the project. The Figma design tool enabled efficient prototyping and a suite of tools which aided in the iterative design process (See figure 4-9).

The technical design in this chapter outlines the applications foundational architecture along with a diagram for the client and server interactions. The technical planning and design aided in identifying compatibility issues with the technology and providing time to solve these issues before the implementation phase.

Choosing the component library Shadcn, along with the Tailwind CSS framework helped maintain consistency throughout the application while also reducing the complexity of the implementation phase.

Ultimately, the design phase provided a guide for the development of the decentralized content management system. This was done through planning and preparing for the implementation phase.

Chapter 5. Implementation

5.1 Introduction

The application developed for this project is a CMS that stores data in a decentralized database. The purpose of this section is to document the process of developing the application. The application was built using Next JS as the foundation, which provides both front-end and back-end functionality, enabling the UI and API to coexist in one monolith codebase.

Gun JS was used for the database, which is a distributed, peer to peer, decentralized graph database. It works by caching data across multiple peers in a network where no one peer has centralized authority over the data. Despite Gun JSs graph functionality, these features are not used in the application.

For decentralized user authentication, the MetaMask SDK is utilized. This works by connecting the users MetaMask wallet retrieving their cryptographic signature for authentication.

The design of the application uses the Shadcn component library in combination with the Tailwind CSS framework. Additionally, the TipTap framework was used to develop the rich text editor component. For storing data on the client, the IDB library was used for easy access the browsers IndexedDB. These elements were discussed further in previous chapters (See chapters 3.3, 4.2).

5.2 Scrum Methodology

For the management of this project the scrum agile framework was used for ensuring best development practices. The implementation phase of this project consisted of seven sprints. A sprint is a two-week period where a list of tasks or tickets would be created and put into a prioritized list named a backlog. Each task in the backlog would be assigned a story point value.

A story point is a representation of how difficult a specific task would be. Minor bug fixes would have a low story point while a larger feature would have a higher point value. The points help manage workload over a sprint and prevents tasks being carried over to the next sprint.

During each sprint weekly meetings were organised with the project supervisor. This was done so feedback and project guidance could be given. At the start of each sprint a backlog refinement would be done where tasks are added and story pointed. Once the backlog has been refined tasks are then selected for the upcoming sprint. Any tasks that are not complete within the sprint are carried over to the next sprint.

In this project the backlog was managed using a Trello board. This gave a visualisation of the backlog and enabled tracking the state of a task. Additionally, tasks were also tracked in the personal reflection journal and in todos.txt in the codebase.

The benefits of the scrum methodology are seen in the speed and quality of development. Managing tasks in a backlog ensures consistent development and productivity which causes faster features being added to the code base. Also, with consistent feedback from the supervisor a high level of quality is maintained throughout the development process.

5.3 Development environment

For the development environment of this application visual studio code was utilized. Additionally, for testing the API both the Thunder client and Postman rest clients were utilized. By default, Gun JS

is configured without any peers. To solve this, the Gun JS developers provided a public Heroku server peer that any Gun JS instance can connect to. For the development of this application this public peer was used alongside a local peer running from local host.

The workflow used for version control was with Git and GitHub. For simplicity after any new feature or bug fix was added a commit would be made directly to the main branch. The name of each commit would be related to the task being completed from the task backlog.

5.4 Demo application

Create a New Collection

Collection Name:

Properties

<input type="text" value="string value"/>	<input type="text" value="String"/>	<input type="button" value="Remove"/>
<input type="text" value="number value"/>	<input type="text" value="Number"/>	<input type="button" value="Remove"/>
<input type="text" value="boolean value"/>	<input type="text" value="Boolean"/>	<input type="button" value="Remove"/>
<input type="button" value="Add Property"/>		
<input type="button" value="Create Collection"/>		

Existing Collections

- car
 - make (string)
 - has_insurance (boolean)
 - year (number)

Items

```
make: BMW
has_insurance: true
year: 2002
make: Audi
has_insurance: false
year: 2024
```

Add an Item to car

make (string):	<input type="text" value="Toyota"/>
has_insurance (boolean):	<input type="text" value="true"/>
year (number):	<input type="text" value="2015"/>
<input type="button" value="Add Item"/>	

Figure 5-1 Pre sprint demo CMS.

Before the first sprint, a demo application was created (See figure 5-1). This demo was built using Gun JS, TypeScript, and React. There was no styling in this application, but custom collections could

be created with their own properties and stored in the Gun JS database. Additionally, single entries in a collection could be retrieved alongside the entire collection.

```
import Gun from 'gun'
import { NextResponse } from 'next/server'
import { Acknowledgment } from './types'

const gun = Gun(process.env.NEXT_PUBLIC_GUN_URL)

export const POST = async (req: Request, { params }: { params: { data: string } }) => {
  const { data } = await params
  const body = await req.json()
  const ref = gun.get(data)

  const newData = {
    data: body,
    userId: 'admin'
  }

  return new Promise((resolve) => {
    ref.set(newData, (ack: Acknowledgment) => {
      if (ack.err) {
        resolve(NextResponse.json({ message: 'Failed to save data' }));
      } else {
        resolve(NextResponse.json({ message: `Data created: ${String(data)}`, body: newData }));
      }
    });
  });
});

export const GET = async (req: Request, { params }: { params: { data: string } }) => {
  const { data } = await params

  const ref = gun.get(data)

  // fix me later
  const results: any[] = []

  await ref.map().once((res, id) => {
    if (res && res.userId === 'admin') {
      results.push({ id, ...res })
    }
  })

  // Gun needs to wait a second for GET to work
  // this doesnt work because its scoped to inside set time out
  // setTimeout(() => {
  //   return NextResponse.json(results)
  // }, 1000)

  // this is my temp solution to this issue
  await new Promise(resolve => setTimeout(resolve, 1000))

  return NextResponse.json(results)
}
```

Figure 5-2 Pre sprint sample API POST route.

The proof-of-concept code above was moved over to a Next JS codebase. There were no issues with this integration, as the demo was written in vanilla React, which is fully compatible in Next JS. Linting rules were established to ease the development process and maintain a consistent style across the codebase. The biggest feature added in the new codebase before the first sprint was seen above. The proof-of-concept Gun JS code was converted into basic API routes where an arbitrary string could be posted to the database, and all strings could be returned from the database where the user ID (which is hard-coded as admin for this stage) matches the database entry. These routes had no encryption or authentication at the time of creation (See figure 5-2).

The biggest challenge with this was in the GET route. Gun JS offers built-in functions for retrieving and setting data in the database. The map function loops through the data in each Gun JS reference. A Gun JS reference is just the name-value for where data is stored in the database. Gun JS is a graph

database; that's why it needs its own specific map function, as opposed to the built-in map function that works on arrays. The once built-in method gets data once and takes in a callback where data can be transformed or returned. By combining map and once, data can be retrieved from the Gun JS reference and pushed to a results array variable, which is sent to the client. Gun JS requires a timeout promise to block the thread while the results variable is assigned before being returned to the client. This can be seen at the bottom of the code and is a pattern that is repeated throughout the API code.

```
import { Model } from "../types"
import { NextResponse } from "next/server"
import fs from 'fs'
import path from 'path'

export const POST = async (req: Request) => {
  try {
    const filePath = path.join(process.cwd(), 'public', 'models.json')
    const newModel: Model = await req.json()
    const modelsJson = await JSON.parse(fs.readFileSync(filePath, 'utf-8'))

    modelsJson.push(newModel)

    fs.writeFileSync(filePath, JSON.stringify(modelsJson))

    console.log(newModel)

    return NextResponse.json({ message: 'added to models success' })
  }
  catch (error) {
    return NextResponse.json({ error: error })
  }
}
```

Figure 5-3 Model save route.

During this phase, a model save route was added. When the user defines a collection from the UI with custom properties, these models must persist between refreshes so the developer using the CMS can add rows that match their own defined models with strict type validation. This route takes a model, stringifies it, and then pushes it to the public directory in a file named models.json (See figure 5-3).

5.5 Sprint 1

5.5.1 Goal

For the first sprint, the top priority was to finish the data model, as it would inform the entire development process alongside the completion of the design process. The only programming task in this sprint was the inclusion of cleaning up the API response body by removing redundant properties.

5.5.2 Item 1

The primary goal of this sprint was to finalize the data model. This item was the top priority, as it would inform the entire development phase. Using Figma and draw.io data, several data models were designed. These can be seen above (See figure 4-1, 4-2, 4-3).

5.5.3 Item 2

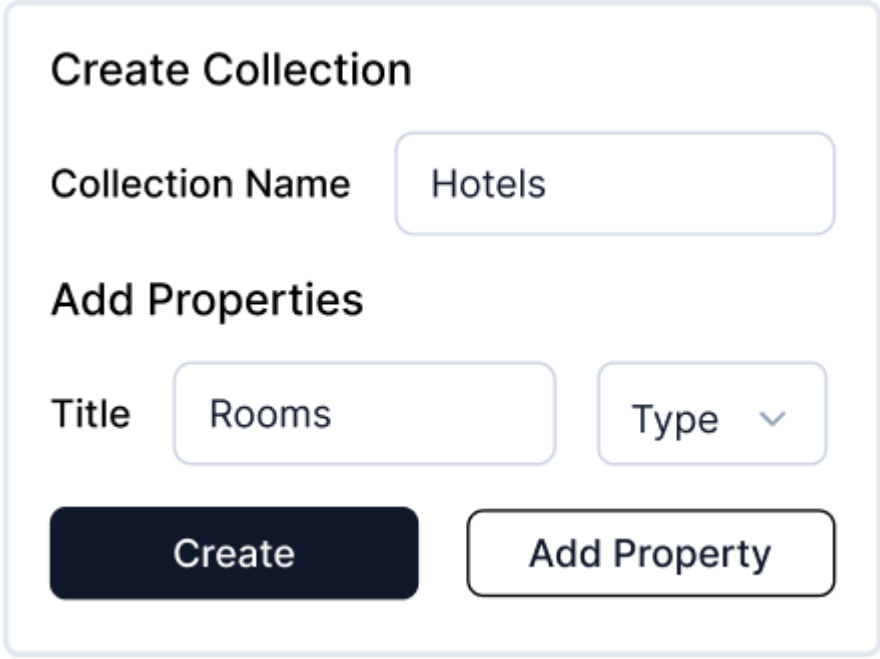


Figure 5-4 Create collection component.

Alongside Item 1, the final iteration of the UI designs had to be completed. Before starting this item, Shadcn components had been selected from their documentation, and a wireframe had been created for the structure of the application. The work of this item involved placing the selected components into the wireframe and tweaking the final iteration. Additionally, a custom component for the collection creator was created (See figure 5-4).

5.5.4 Item 3

```
export const cleanResponse = (response: Item[]) => {  
  return response.map(item => {  
    const { _, ...newObject } = item  
    return newObject  
  })  
}
```

Figure 5-5 Clean response function.

The final item in this sprint was a small bug/quality of life fix for the Gun JS get route. As Gun JS is a graph database, the response returned from the API had boilerplate code with pointers to other entries in the graph database. Since the CMS did not need graph functionality, this was redundant and had to be cleared from the response. The response array would be passed into the function, and all its graph pointers were stored in the `_` property. The response would be mapped, and the `_` property would be dropped. This made the response more human-readable and usable throughout the rest of the codebase (See figure 5-5).

5.6 Sprint 2

5.6.1 Goal

Shadcn UI framework was configured in the project. Another task taken on in this sprint was getting user authentication working with the MetaMask SDK. This also led into the next task of writing first-time project setup code for developers using the CMS. The API routes were expanded from the demo in this sprint as well, and the creation of tokens was started in this sprint but carried over to the next sprint.

5.6.2 Item 1

```
// https://metamask.io/news/developers/how-to-implement-metamask-sdk-with-nextjs/

import { useSDK } from '@metamask/sdk-react'

export const ConnectWalletButton = () => {

  const { sdk, connected, account } = useSDK()

  const connect = async () => {
    try {
      await sdk?.connect()
    } catch (err) {
      console.warn(`No accounts found`, err)
    }
  }

  const disconnect = () => {
    if (sdk) {
      sdk.terminate()
    }
  }

  return (
    <div className='relative'>
      {connected ? (
        <>connected
        <button onClick={() => disconnect()}>disconnect</button>
        {account}
      </>
      ) : (
        <button onClick={() => connect()}>connect wallet</button>
      )}
    </div>
  )
}
```

Figure 5-6 Connect wallet component.

The first item in this sprint was to get user authentication with the MetaMask SDK. MetaMask provides a React SDK that connects the browser to the user's MetaMask wallet to enable decentralized user authentication with the wallet's private key. The private key in MetaMask is what's used for signing and verifying transactions of cryptocurrency. A ConnectWallet component was made that simply calls the connect function, which opens the MetaMask Chrome extension and asks for the user's password. Any component using this SDK must be wrapped in the MetaMask provider component (See figure 5-6).

5.6.3 Item 2

```
import { NextResponse } from "next/server"
import fs from 'fs'
import path from 'path'

export const POST = async (req: Request) => {
  try {
    const filePath = path.join(process.cwd(), 'public', 'auth.json')

    // // add type later
    // const params = await req.json

    // check if the auth.json file exists if not means it the first time logging in
    if (!fs.existsSync(filePath)) {
      fs.writeFileSync(filePath, JSON.stringify([]))

      console.log('first time logging in')
    }

    // const isLoggedInArray = JSON.parse(fs.readFileSync(filePath, 'utf-8'))

    // if (isLoggedInArray.length === 0) {

    //   isLoggedInArray.push({ firstTimeLogin: true })

    //   fs.writeFileSync(filePath, JSON.stringify(isLoggedInArray))
    //   return NextResponse.json({ message: 'First-time login set to true' })
    // }

    return NextResponse.json({ message: 'fdjlsfjdl' })
  } catch (error) {
    return error
  }
}
```

Figure 5-7 Auth user route.

Additionally, the first-time login route was added to the codebase. This route is designed for the first-time setup of the application. The current code stores the user's public wallet address from the MetaMask SDK and saves it in the public directory. In future sprints, this route will be used for generating the public and private RSA keys / API key for authentication using the crypto / JWT libraries. Additionally, this route will create the application's env file, which is needed for the

application to run properly. These features were developed at the end of the sprint and can be seen below (See figure 5-7).

```
const createToken = (sessionData: string, privateKey: string) => {
  if (privateKey) {
    return jwt.sign(sessionData, privateKey, { algorithm: 'RS256' })
  }
}

const generateRSAKeys = async () => {
  try {
    // use crypto to generate public / private keys
    const rsaKeyPair = await new Promise((resolve, reject) => {
      generateKeyPair(
        'rsa',
        {
          modulusLength: 2048,
          publicKeyEncoding: { type: 'spki', format: 'pem' },
          privateKeyEncoding: { type: 'pkcs8', format: 'pem' }
        },
        // fix any later
        (err: any, publicKey: any, privateKey: any) => {
          if (err) reject(err)
          else resolve({ publicKey, privateKey })
        }
      )
    })

    return rsaKeyPair
  } catch (error) {
    console.error(error)
  }
}
```

Figure 5-8 Create token & generateRSAkey.

```
const createEnvFile = (rsaKeyPair: any, walletId: string) => {
  const fileName = '.env'
  const filePath = path.resolve(fileName)

  const initialSession = startSession(walletId)
  const apiToken = createToken(initialSession, rsaKeyPair.privateKey)

  if (!fs.existsSync(filePath)) {
    const defaultContent = `PUBLIC_RSA_KEY="${rsaKeyPair.publicKey}"\nPRIVATE_RSA_KEY="${rsaKeyPair.privateKey}"\nNEXT_PUBLIC_HOSTING_URL="http://localhost:3000"\nNEXT_PUBLIC_GUN_URL="https://gun-marshattan.herokuapp.com/gun"\nPUBLIC_API_TOKEN="${apiToken}"`
    fs.writeFileSync(filePath, defaultContent)
    console.log(`${fileName} created successfully.`)
  } else {
    console.error(`${fileName} already exists.`)
  }
}
```

Figure 5-9 Create ENV function.

```

export const POST = async (req: Request, res: NextApiResponse) => {
  try {
    const filePath = path.join(process.cwd(), 'public', 'auth.json')

    const { walletId } = await req.json()

    // check if the auth.json file exists if not means it the first time logging in
    if (!fs.existsSync(filePath)) {
      fs.writeFileSync(filePath, JSON.stringify([ { 'firstTimeLogin': true, 'walletId': walletId } ]))

      const rsaKeyPair = await generateRSAKeys()

      await createEnvFile(rsaKeyPair, walletId)

      return NextResponse.json({ message: 'First-time login set to true' })
    }

    return NextResponse.json({ message: 'Not first-time logging in' })
  } catch (error) {
    return NextResponse.json({ message: 'An Error occurred', error: error })
  }
}

```

Figure 5-10 Updated user auth route for first time setup.

Starting from the top, the first function defined is createToken. This helper function takes in a private RSA key and a session data string, which includes the date set up and public wallet address and generates an API token using the JWT library (See figure 5-8).

The generateRSAkey function uses the crypto library to generate the public and private RSA keys, which are used for encrypting and decrypting the user's data in the Gun JS database. The key pair generation is a promise that calls the generateKeyPair function. This function takes in a config with the public and private key encoding (spki and pkcs8), and both are using the PEM format, which prefixes the key with a "start of key" string. This promise is awaited to resolve, then returned. There is also a catch block within the promise callback to log any errors while generating the key pairs.

The createEnvFile function takes in the key pairs after they have been generated and the wallet ID. This function starts a session and then uses that to create an API token. When this is done, the function checks if an .env file has been created. If the file has not been created, it will create the .env file and append the generated keys and the Gun JS public hosting URL to the file (See figure 5-9).

The final POST route is what's called when the user logs in. If the auth.json file does not exist, it means it's the user's first time logging in and entering the project setup phase. In this case, the defined functions above are called, generating the .env file with its keys and the Gun JS hosting URL (See figure 5-10).

This is necessary to ensure the security of the decentralized database. Without the starting hosting URL, the database cannot function as a decentralized database, as the data would only be stored in the local cache, essentially making it a centralized database. Also, having tamper-proof encryption is critically important to ensure data safety and user privacy.

5.7 Sprint 3

5.7.1 Goal

Many tasks were taken on in this sprint. The most important items were the inclusion of encryption and security in the application. The API was the primary focus completing the collection CRUD routes, along with the single-valued routes. The save API response helper route would be created as well. Additionally, this sprint marked the start of the UI development phase.

5.7.2 Item 1

```
export const encryptData = (data: Item | User) => {
  const buffer = Buffer.from(JSON.stringify(data))
  const encrypted = crypto.publicEncrypt(process.env.PUBLIC_RSA_KEY as string, buffer)

  return encrypted.toString('base64')
}

export const generateSignature = (data: Item | User) => {
  const sign = crypto.createSign('SHA256')
  sign.update(JSON.stringify(data))

  const signBuffer = sign.sign(process.env.PRIVATE_RSA_KEY as string)

  return signBuffer.toString('base64')
}

export const decryptData = (encryptedData: string): Item | User | Single => {
  const buffer = Buffer.from(encryptedData, 'base64')
  const decrypted = crypto.privateDecrypt(process.env.PRIVATE_RSA_KEY as string, buffer)
  return JSON.parse(decrypted.toString('utf-8'))
}
```

Figure 5-11 Encryption decryption and data signature functions.

The first item in this sprint was the inclusion of encryption, decryption, and data signatures in the API. This is vital to the overall security of the CMS and database. The first function created for this was the `encryptData` function. This takes in a data variable—any item about to be added to the database—then converts it to a buffer using the `crypto` library after stringifying the data. It is then encrypted using the generated public RSA encryption key and returned from the function as a string. This function is called in the API routes before data is added to the Gun JS database.

The next function created in this sprint was the `decryptData` function. This function takes in an encrypted string, then converts it to a buffer data type. Using the `crypto` library and the private RSA key, the data is decrypted and then parsed using the `JSON.parse` method to convert it back into a regular TypeScript object.

Additionally, the `generateSignature` function was created as part of this sprint. The purpose of this function is to verify the integrity of data and ensure it has not been tampered with. The function takes in data that will be added to the database, then uses the `crypto` library to create a SHA-256

signature of the data. It updates the signature using a stringified version of the data. A sign buffer is created using the private key, and this is returned from the function as a string.

Both the encrypted data and the data signature require the private key to be decrypted. Once this is done, both items are compared. If they do not match, it means the data has been tampered with by an unauthorized user (See figure 5-11).

```
export const POST = async (req: Request, { params }: { params: { modelId: string } }) => {
  const authHeader = await req.headers.get('Authorization')

  // verify token
  const checkToken = await authorisationMiddleware(authHeader)
  if(checkToken) return checkToken

  const { modelId } = await params
  const body = await req.json()
  const ref = gun.get(modelId)

  // generate an data integrity signiture / encrypt data
  const signature = generateSignature(body)
  const encryptedData = encryptData(body)

  const newData = {
    encryptedData,
    signature,
    id: generateRowId()
  }

  ref.set(newData, (ack: Acknowledgment) => {
    if (ack.err) {
      return NextResponse.json({ message: 'Failed to save data' })
    }
  })

  return NextResponse.json({ message: `Data created`, body: newData })
}
```

Figure 5-12 Add new row route with encryption & data signatures.

Above is an example of the encryptData and generateSignature functions being used in the create route. The route takes the modelId, which is a unique string used for storing data in the database using the title as an ID. This is done by creating a reference to the Gun JS database with this unique ID. Then, the route gets the body variable, which is the data that will be added to the database. This data is then encrypted, and a data signature is generated. These are put into a newData variable, along with an id property that generates a unique ID for referencing this specific entry in the database. Finally, using the reference and the set method, this new data is added to the database with a callback for error handling (See figure 5-12).

5.7.3 Item 2

```

const verifyToken = (token: string) => {
  try {
    const publicKey = process.env.PUBLIC_RSA_KEY

    if (!token) {
      console.error('Token is missing!')
      return false
    }

    if (!publicKey) {
      console.error('Public key is missing!')
      return false
    }

    const decoded = jwt.verify(token, publicKey, { algorithms: ['RS256'] })

    if (decoded) {
      return true
    }

    return false
  } catch (error) {
    console.error('Error verifying the token:', error)
    return false
  }
}

```

Figure 5-13 Verify token function.

```

// this technically is not middleware but the way next handles middleware
// is that it creates an edge environment and you define the paths where the middleware runs
// unfortunately you cant use node packages in edge envs
// its easier to just make this function and treat it like middleware in resource routes
export const authorisationMiddleWare = (authHeader: string | null) => {
  if (!authHeader) {
    return NextResponse.json({ message: 'no auth token present', ok: false }, { status: 401 })
  }

  // get the token
  const token = authHeader.split(' ')[1]

  if (!verifyToken(token)) {
    return NextResponse.json({ message: 'error fetching, invalid token', ok: false }, { status: 401 })
  }
}

```

Figure 5-14 Authorisation middleware.

The next item completed in this sprint was the inclusion of authenticated routes. For each route, an API token must be present to authenticate the user hitting the endpoint. It's worth noting that while this is referred to as middleware throughout the project, it technically is not middleware. In Next JS,

when middleware is created, it runs in an edge environment as opposed to a Node environment like the resource routes. This causes issues, as the JWT and crypto packages cannot be run in edge environments—only in Node.

The authorization middleware function takes in an AUTH header and checks that it is present and that it passes the `verifyToken` function.

The `verifyToken` function works by taking the token and decoding it using JWT. Verifying it with the public key. If the decoded variable is returned, the function will return `true`, as the token is valid. If the token is invalid, this variable will not be returned, and the function will return `false`. This function also includes a check to ensure that both a token and a public key are provided (See figure 5-13, 5-14).

5.7.4 Item 3

```
export const GET = async (req: Request, { params }: { params: { modelId: string, rowId: string } }) => {
  const authHeader = await req.headers.get('Authorization')

  // verify token
  const checkToken = await authorisationMiddleWare(authHeader)
  if (checkToken) return checkToken

  const { modelId, rowId } = await params

  const ref = gun.get(modelId)
  const results: Item[] = []

  await ref.map().once((res) => {
    if (res && res.id === rowId) {
      const decryptedData = decryptData(res.encryptedData)
      const isValid = verifySignature(decryptedData, res.signiture)

      // if the signature is valid it means the data has not been tampered with outside of the api
      if (isValid) {
        results.push({ ...decryptedData, id: res.id })
      }
      else {
        // alert the user to the fact that unauth data has been altered / add in roll back feature
        console.error('unauth user altered data start rollback')
      }
    }
  })

  // Gun needs to wait a second for GET to work
  // this doesnt work because its scoped to inside set time out
  // setTimeout(() => {
  //   return NextResponse.json(results)
  // }, 1000)

  // this is my temp solution to this issue
  await new Promise(resolve => setTimeout(resolve, 1000))

  return NextResponse.json(cleanResponse(results))
}
```

Figure 5-15 Get single row route with data signature verification.

In this item in the sprint, the other resource routes (get single, update, delete) were added to the collections API. Additionally, the data signature verification code was added as part of this item. The

first one implemented was the get single route, which can be seen above. This route is essentially the same as the get all route, but a rowId is passed to the route with the modelId.

After checking the auth token, a reference to the Gun JS database is made using the modelId. Using the Gun map and once methods, all the data under this reference is checked to see if the id matches the rowId passed to the route. If the IDs match, the data is decrypted, and the validity of the signature is checked. If the signature is valid, that row is then pushed to a results array, which is returned at the end of the code (See figure 5-15).

```
export const PUT = async (req: Request, { params }: { params: { modelId: string, rowId: string } }) => {
  try {
    const authHeader = await req.headers.get('Authorization')

    // verify token
    const checkToken = await authorisationMiddleware(authHeader)
    if (checkToken) return checkToken

    const body = await req.json()

    // Extract modelId and rowId from the params
    const { modelId, rowId } = await params
    const ref = gun.get(modelId)

    if (!body || !modelId) {
      return NextResponse.json({ message: 'invalid params', ok: false }, { status: 400 })
    }

    // fix any later
    const results: any = {}

    await ref.map().once((res) => {
      if (res && res.encryptedData) {
        const decryptedData = decryptData(res.encryptedData)
        const isValid = verifySignature(decryptedData, res.signature)

        // if the current entry is valid and the id matches the row id param
        // create a new body and add it as a property to results
        if (isValid && res.id === rowId) {
          const combinedBody = { ...decryptedData, ...body }

          const newBody = {
            id: rowId,
            encryptedData: encryptData(combinedBody),
            signature: generateSignature(combinedBody),
          }

          results[getGunEntryId(res)] = newBody
        }

        // if the entry is valid but does not match
        // dont edit it and add it to results
        else if (isValid) {
          results[getGunEntryId(res)] = res
        }

        else {
          // alert the user to the fact that unauth data has been altered / add in roll back feature
          console.error('unauth user altered data start rollback')
        }
      }
    })

    // this is my temp solution to this issue
    await new Promise(resolve => setTimeout(resolve, 1000))

    ref.put(results, (ack: Acknowledgment) => {
      if (ack.err) {
        console.error(ack.err)
        return NextResponse.json({ message: 'Failed to update data', ok: false }, { status: 500 })
      }
    })

    return NextResponse.json({ body: cleanResponse(results), message: 'Successfully updated row', ok: true }, { status: 200 })
  }
}
```

Figure 5-16 PUT route in API

The next route added was the update (or PUT) route. This works similarly to the create and get single routes. After the authentication middleware, the route gets a reference to the Gun JS database. There is also a check to ensure that a body and modelId are passed to the route.

It uses the map and once methods to decrypt the data and verify that all the data signatures are valid. If the data is valid and matches the rowId, it will create a combinedBody variable. This is a combination of the updated body passed by the user and the decrypted data from the database. This is then encrypted, and a new signature is generated in the newBody variable, along with the rowId.

This is then added to the results variable using the getGunEntryId function, which returns the pointer value of the entry in the database. While this pointer isn't often used in the application (since graph functionality isn't needed), it is necessary in this case.

If the data in the map function is valid but does not match the rowId, it is also added to the results variable. This is because for the update to work, the whole collection must be updated at once. This can be seen at the bottom of the route with the ref.put method, where the entire reference is overwritten with the results variable.

This is not an optimal solution, but there is no obvious drop in performance. Ideally, this will be updated to a more efficient method in the future (See figure 5-16).

```

export const DELETE = async (req: Request, { params }: { params: { modelId: string, rowId: string } }) => {
  try {
    const authHeader = await req.headers.get('Authorization')

    // verify token
    const checkToken = await authorisationMiddleware(authHeader)
    if (checkToken) return checkToken

    // Extract modelId and rowId from the params
    const { modelId, rowId } = await params
    const ref = gun.get(modelId)

    let rowToDeleteId: string | undefined

    await ref.map().once((res) => {
      if (res && res.encryptedData) {
        const decryptedData = decryptData(res.encryptedData)
        const isValid = verifySignature(decryptedData, res.signature)

        // if the current entry is valid and the id matches the row id param
        // store the row id then delete it later
        if (isValid && res.id === rowId) {
          rowToDeleteId = getGunEntryId(res)
        }
        else if (!isValid) {
          // alert the user to the fact that unauth data has been altered / add in roll back feature
          console.error('unauth user altered data start rollback')
        }
      }
    })

    // Gun needs to wait a second for GET to work
    // this doesnt work because its scoped to inside set time out
    // setTimeout(() => {
    //   return NextResponse.json(results)
    // }, 1000)

    // this is my temp solution to this issue
    await new Promise(resolve => setTimeout(resolve, 1000))

    if (rowToDeleteId) {
      ref.get(rowToDeleteId).put(null, (ack: Acknowledgment) => {
        if (ack.err) {
          console.error(ack.err)
          return NextResponse.json({ message: 'Failed to delete data', ok: false }, { status: 500 })
        }
      })
    }
    else {
      return NextResponse.json({ message: 'Row not found', ok: false }, { status: 500 })
    }

    return NextResponse.json({ message: 'Successfully deleted row', ok: true }, { status: 200 })
  }
}

```

Figure 5-17 Delete row route.

Similar to the update route, the delete route was more complicated to implement than initially expected. The same pattern of authenticating, getting a reference, and then using the map and once methods was used. In the once callback, the data is decrypted, and the data signature's validity is checked.

If it's valid and the response id matches the row id, the pointer value, in the database for the item to be deleted is assigned to the rowToBeDeletedId variable. At the end of the route, if the

rowToBeDeleted variable is truthy, it will then get the row at the variable's pointer value and update it to be null, deleting the item (See figure 5-17).

5.7.5 Item 4

```
export const POST = async (req: NextRequest) => {
  try {
    const { email, password } = await req.json() as User

    if (!email || !password) {
      return NextResponse.json({ message: 'Invalid params', ok: false }, { status: 400 })
    }

    // validate email / password
    const validEmail = validateEmail(email)
    const validPassword = validatePassword(password)

    if (!validEmail) {
      return NextResponse.json({ message: 'Invalid email', ok: false }, { status: 400 })
    }

    if (!validPassword) {
      return NextResponse.json({ message: 'Invalid password', ok: false }, { status: 400 })
    }

    const body = { email, password }

    // generate an data integrity signature / encrypt data
    const signature = generateSignature(body)
    const encryptedData = encryptData(body)

    const newData = {
      encryptedData,
      signature,
    }

    const ref = gun.get(email)

    let isEmailUnique: undefined | boolean

    // check that the email is unique
    ref.once((res: EncryptedItem) => {
      if (!res) {
        isEmailUnique = true
      }
    })

    await new Promise(resolve => setTimeout(resolve, 1000))

    if (!isEmailUnique) {
      return NextResponse.json({ message: 'Email is not unique', ok: false }, { status: 500 })
    }

    ref.set(newData, (ack: Acknowledgment) => {
      if (ack.err) {
        console.error(ack.err)
        return NextResponse.json({ message: 'Failed to register user', ok: false }, { status: 500 })
      }
    })

    return NextResponse.json({ message: 'User created', token: jwt.sign({ email, password }, process.env.PUBLIC_API_TOKEN), ok: true }, { status: 201 })
  }
}
```

Figure 5-18 API register user route.

Also, in this sprint, user login and registration were added to the API. This is separate from the MetaMask dashboard authentication. This is because we want the API to handle user authentication for apps that use the CMS as a backend. This does not mean they should be authenticated to use the dashboard, only the app that uses the CMS.

Above is the register route. An email and password are passed to the route, and these are validated. If they are invalid, an error is returned from the route. If valid, the data is encrypted, and a data signature is generated. Then, Gun makes a reference to the database using the email. Using map and once, the route checks if the email is unique (i.e., if there is already an entry in the database under the user's email). If the email is unique, the reference uses the set method to add the encrypted email and password to the database. If this is successful, a response is sent to the client with a success message and a JWT token verifying the authenticated user (See figure 5-18).

```

export const POST = async (req: NextRequest) => {
  try {
    const { email, password } = await req.json() as User

    if (!email || !password) {
      return NextResponse.json({ message: 'Invalid params', ok: false }, { status: 400 })
    }

    // validate email / password
    const validEmail = validateEmail(email)
    const validPassword = validatePassword(password)

    if (!validEmail) {
      return NextResponse.json({ message: 'Invalid email', ok: false }, { status: 400 })
    }

    if (!validPassword) {
      return NextResponse.json({ message: 'Invalid password', ok: false }, { status: 400 })
    }

    let correctPasword: undefined | boolean

    gun.get(email).map().once((res: EncryptedItem) => {
      if (res) {
        const decryptedUser = decryptData(res.encryptedData)
        const isValid = verifySigniture(decryptedUser, res.signiture)

        if (isValid && decryptedUser.password === password) {
          correctPasword = true
        }
      }
    })

    await new Promise(resolve => setTimeout(resolve, 1000))

    if (!correctPasword) {
      return NextResponse.json({ message: 'Incorrect password or email', ok: false }, { status: 405 })
    }

    return NextResponse.json({ message: 'User logged in', token: jwt.sign({ email, password }, process.env.PUBLIC_API_TOKEN), ok: true }, { status: 201 })
  } catch (error) {
    console.error(error)
  }
}

```

Figure 5-19 API login user route.

Also, in this item is the login route. This works in an equivalent manner to the registration route. The email and password are passed to the route, then validated. If valid, Gun makes a reference to the email and uses the map and once methods on the reference. In the once method callback, the email-password pair is decrypted, and the signature is checked. If valid and the decrypted password matches, it means the user is authentic, and the correctPassword variable is set to true.

At the end of the route, if the correctPassword is true, the route sends a positive response to the client with a JWT token (See figure 5-19).

5.7.6 Item 5

```
export const POST = async (req: NextRequest) => {
  try {
    const { url, status } = await req.json() as StatusFromAPI

    const filePath = path.join(process.cwd(), 'public', 'response.json')

    if (!status || !url) {
      return NextResponse.json({ message: 'Invalid url or status params', ok: false }, { status: 500 })
    }

    const currentResponses = JSON.parse(fs.readFileSync(filePath, 'utf-8'))

    fs.writeFileSync(filePath, JSON.stringify([...currentResponses, { url, status }]))

    return NextResponse.json({ message: 'Saved API response status', ok: true }, { status: 201 })
  } catch (error) {
    console.error(error)
  }
}

export const GET = async (req: NextRequest) => {
  try {
    const filePath = path.join(process.cwd(), 'public', 'response.json')

    const response = JSON.parse(fs.readFileSync(filePath, 'utf-8'))

    return NextResponse.json({ data: response, ok: true }, { status: 200 })
  } catch (error) {
    console.error(error)
    return NextResponse.json({ message: 'Error reading file', ok: false }, { status: 500 })
  }
}
```

Figure 5-20 Save response status route.

One of the most key features found during the requirements phase was the inclusion of an API status page. This page shows all the API calls made and their status codes. This aids developers when using the CMS, as they can see where API calls are failing and fix these issues.

The routes above take in a StatusFromAPI type, which has a URL property and a status property, and pushes them to the response.json file in the public directory. There is another GET route that simply returns all the statuses from the file as well (See figure 5-20).

5.7.7 Item 6

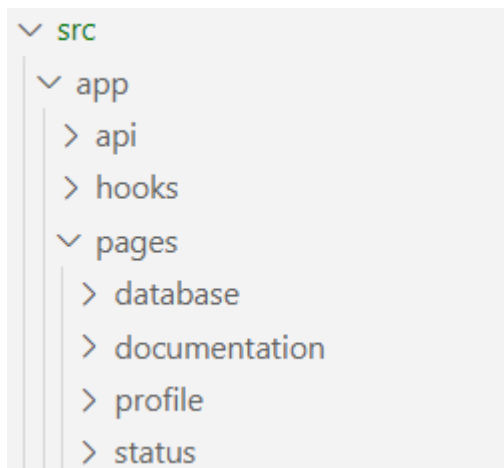


Figure 5-21 Src directory.

The API work was now complete, aside from bug fixes and improvements. This item marks the start of the UI development phase of the implementation process. In the Next JS app directory, the pages directory was added, with subdirectories for database, documentation, profile, and status. Due to Next's directory-based routing, these all correspond to pages within the application.

Additionally, several Shadcn components were added to the application. These can be seen in the UI directory and include Card, Separator, and Button (Figure 5-21).

```
useEffect(() => {
  account && setIsLoggedIn(true)
}, [account])

if(connected) return

return (
  <div className='relative'>
    <div className='flex items-center justify-center min-h-screen'>
      <Card className='w-64'>
        <CardHeader>
          <CardTitle>Connect Wallet</CardTitle>
          <CardDescription>Metamask account and extension are required</CardDescription>
        </CardHeader>
        <CardContent>
          <Button onClick={() => connect()}>
            <Mail /> Login with Metamask
          </Button>
        </CardContent>
      </Card>
    </div>
  </div>
)
```

Figure 5-22 Code for MetaMask login button.

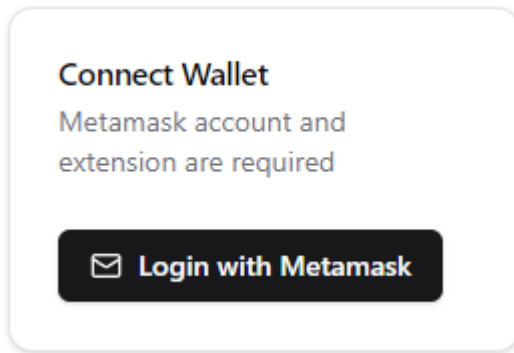


Figure 5-23 MetaMask login component UI.

Using the Card and Button components, the ConnectWalletButton component had styles added. The component works very simply—it uses the built-in Card component for styling and checks if an account variable exists in a useEffect. If that variable exists, it means the user has been authenticated with the MetaMask SDK. If they are authenticated, they are redirected to the database page (See figure 5-22, 5-23).

5.7.8 Item 7

```
const Sidebar = () => {

  const pathName = usePathname()
  const router = useRouter()

  return (
    <div className='h-screen w-20 flex flex-col items-center bg-[#F8FAFC] border-r border-[#E2E8F0] text-white py-4 space-y-5'>
      <button onClick={() => router.push('/database')} className='p-2 rounded-lg'>
        <Database className={`w-6 h-6 ${pathName === '/database' ? 'text-black' : 'text-[#94A3B8]'}`} />
      </button>

      <button onClick={() => router.push('/documentation')} className='p-2 rounded-lg'>
        <Code2 className={`w-6 h-6 ${pathName === '/documentation' ? 'text-black' : 'text-[#94A3B8]'}`} />
      </button>

      <button onClick={() => router.push('/status')} className='p-2 rounded-lg'>
        <Activity className={`w-6 h-6 ${pathName === '/status' ? 'text-black' : 'text-[#94A3B8]'}`} />
      </button>

      <button onClick={() => router.push('/profile')} className='p-2 rounded-lg'>
        <User className={`w-6 h-6 ${pathName === '/profile' ? 'text-black' : 'text-[#94A3B8]'}`} />
      </button>
    </div>
  )
}
```

Figure 5-24 Side navigation bar code.



Figure 5-25 Side navigation bar.

With the routes for the pages of the CMS defined, a side navigation bar was added for switching between the pages. It was created by using a div with multiple buttons for each route. When a button is pressed, the `useRouter` hook is used to change the path in the app to the selected route. Additionally, the `usePathname` hook and conditional styling with the ternary operator are used to highlight the currently selected route in black, while the other buttons are grey (See figure 5-24, 5-25).

```
const useAuthentication = () => {  
  const { isLoggedIn } = useAuth()  
  const router = useRouter()  
  
  useEffect(() => {  
    !isLoggedIn && router.push('/')  
  })  
}  
  
export default useAuthentication
```

Figure 5-26 Custom hook for user authentication.

```
import { AuthContextInterface } from '@types'
import { createContext, useContext, useState, ReactNode } from 'react'

const AuthContext = createContext<AuthContextInterface | undefined>(undefined)

export const AuthProvider = ({ children }: { children: ReactNode }) => {
  const [isLoggedIn, setIsLoggedIn] = useState<boolean>(false)
  const [walletAddress, setWalletAddress] = useState<string | null>(null)

  return (
    <AuthContext.Provider value={{ isLoggedIn, setIsLoggedIn, walletAddress, setWalletAddress }}>
      {children}
    </AuthContext.Provider>
  )
}

export const useAuth = () => {
  const context = useContext(AuthContext)
  if (!context) {
    throw new Error('No context provided')
  }
  return context
}
```

Figure 5-27 React context for auth data.

Additionally, the entire UI section of the application is wrapped in an AuthContext, which uses a useAuthentication custom hook that reads the context to determine whether the user has been authenticated. If the user has not completed decentralized authentication through MetaMask, the hook reroutes them to the login page. The AuthContext has two state variables: one for whether the user is logged in and another for the user's wallet address (See figure 5-26, 5-27).

5.7.9 Item 8

API / Logs

Filter by endpoint

<input type="checkbox"/> STATUS	URL	CREATED_AT
<input type="checkbox"/> 200	http://localhost:3000/api/collections/Todos-349528	26/3/2025, 11:36:12
<input type="checkbox"/> 200	http://localhost:3000/api/collections/Todos-349528	26/3/2025, 11:36:11
<input type="checkbox"/> 200	http://localhost:3000/api/collections/Todos-349528	26/3/2025, 11:35:57
<input type="checkbox"/> 200	http://localhost:3000/api/collections/Todos-349528	26/3/2025, 11:35:56
<input type="checkbox"/> 201	http://localhost:3000/api/collections/Todos-349528	26/3/2025, 11:35:10
<input type="checkbox"/> 500	http://localhost:3000/api/collections/Todos-349528	26/3/2025, 11:35:02
<input type="checkbox"/> 201	http://localhost:3000/api/collections/Todos-349528	26/3/2025, 11:35:01
<input type="checkbox"/> 200	http://localhost:3000/api/collections/Todos-349528	26/3/2025, 11:34:03
<input type="checkbox"/> 200	http://localhost:3000/api/collections/Todos-349528	26/3/2025, 11:34:02
<input type="checkbox"/> 500	http://localhost:3000/api/collections/Todos-349528	26/3/2025, 11:32:02
<input type="checkbox"/> 201	http://localhost:3000/api/collections/Todos-349528	26/3/2025, 11:32:02
<input type="checkbox"/> 200	http://localhost:3000/api/collections/Todos-349528	26/3/2025, 11:30:50
<input type="checkbox"/> 200	http://localhost:3000/api/collections/Todos-349528	26/3/2025, 11:30:48

Figure 5-28 Status table.

```

const columns: ColumnDef<StatusFromAPI>[] = [
  {
    id: 'select',
    header: ({ table }) => (
      <Checkbox
        checked={
          table.getAllPageRowsSelected() ||
          (table.getIsSomePageRowsSelected() && 'indeterminate')
        }
        onCheckedChange={(value) => table.toggleAllPageRowsSelected(!value)}
        aria-label='Select all'
      />
    ),
    cell: ({ row }) => (
      <Checkbox
        checked={row.getIsSelected()}
        onCheckedChange={(value) => row.toggleSelected(!value)}
        aria-label='Select row'
      />
    ),
  },
  {
    accessorKey: 'status',
    header: () => <div className='w-[-1000px]'>STATUS</div>,
    cell: ({ row }) => {
      const status = row.getValue('status') as string
      const colour = `${status.toString()[0]} === '2' ? 'text-green-500' : 'text-red-500' font-medium`

      return <div className={colour}>{status}</div>
    },
  },
  {
    accessorKey: 'url',
    header: 'URL',
    cell: ({ row }) => (
      <div>{row.getValue('url')}</div>
    ),
  },
  {
    accessorKey: 'createdAt',
    header: () => <div className='text-right mr-10'>CREATED_AT</div>,
    cell: ({ row }) => (
      <div className='text-right'>{row.getValue('createdAt')}</div>
    ),
  },
]

```

Figure 5-29 Shadcn config for status data table.

The API status page was added to the application. This was one of the features discovered during the requirements phase. Users can see their API requests and their status codes. This component was built using the Shadcn DataTable component. This is a prebuilt component that has filtering functionality built into its tables.

The Table component requires a columns config object. The config requires a unique accessorKey, a header, and an optional cell property. Above, you can see the first property in the config is for including a checkbox to select rows in the table. After that, there is the status column, which holds the status code and dynamically changes colour based on that code. Using a ternary, if the status is 200, it will be green; otherwise, it will be red.

The last two columns are the URL and createdAt properties, which hold the API URL and the time the API request was made, respectively.

5.8 Sprint 4

5.8.1 Goal

The first item in this sprint was the creation of the database page, which included making the `CollectionTable` and `InnerSidebar` components. CRUD functionality from the UI was implemented during this sprint. The User's page was also added, along with the Documentation page. Refactoring of the singles API was necessary and completed during this sprint.

5.8.2 Item 1

Collections / Todos

Add New Row

Update Row

Delete Row

id	body	isDone
<input checked="" type="radio"/> c6fec8e0-cc46-402f-a6b6-c774ea44196c	test add from todos app	false
<input type="radio"/> 6f0c328a-efd1-46f9-981a-38fa9f0bf2c2	new todo again	false
<input type="radio"/> 78d1f46c-9d6f-47b4-9f3e-93fbfa34191	show console	false
<input type="radio"/> 27088b57-3fb9-4cae-9e9c-d3cbfd79cbc5	new todo to do	false
<input type="radio"/> 98beeb28-e07e-4824-85fb-c733391a368e	body	true
<input type="radio"/> 8f2b3ce9-d4f1-43cb-ad54-a761d28ba682	new todo	false
<input type="radio"/> 30163aa6-a5e5-4bc0-8b6e-c11fb3b4a286	this actually is the new one	true

Figure 5-30 Database page with collection table and CRUD buttons.

```
const generateColumns = (properties: Property[]): ColumnDef<any>[] => {
  return properties.map((property) => {
    return {
      accessorKey: property.name,
      header: property.name,
      cell: ({ row }) => {
        const value: string | number | boolean = row.getValue(property.name)

        if (property.type === 'boolean') {
          return <div>{value ? 'true' : 'false'}</div>
        }

        return <div>{value}</div>
      },
    },
  })
}
```

Figure 5-31 Generate columns function for collection table.

The first item in this sprint was the development of the collections table. This table needed to show all the entries in each collection with their correct properties, while also being able to perform full CRUD from the UI with the correct validation for a user-defined collection. The most challenging part of this was getting the Shadcn data table to dynamically generate columns. Multiple attempts were made, but the above is the current and most simple implementation. It loops through a properties array and returns them as a `ColumnDef` with a properties array (See figure 5-31).

```

useEffect(() => {
  reset()

  const handleGetAllRows = async () => {
    const allData = await getAllCollectionRows(selectedModel.modelId)

    if (allData) {
      setData(allData)
    }
    else {
      setData([])
    }
  }

  handleGetAllRows()

  return () => {
    reset()
  }
}, [selectedModel])

```

Figure 5-32 API request to get all rows for a selected collection.

Before the table is rendered, the UI uses a `useEffect` hook to get all rows for the selected collection. If the `getAllCollectionRows` function's return value is truthy, the data state variable is assigned the response (See figure 5-32).

```

const [columnFilters, setColumnFilters] = useState<ColumnFiltersState>([])

const columns = generateColumns(selectedModel.properties)
const table = useReactTable({
  data: data || [],
  columns,
  getCoreRowModel: getCoreRowModel(),
  getSortedRowModel: getSortedRowModel(),
  onColumnFiltersChange: setColumnFilters,
  getFilteredRowModel: getFilteredRowModel(),
  state: {
    columnFilters,
  },
})

```

Figure 5-33 `useReactTable` hook for generating dynamic data table for collections.

After the columns are generated using the selected model's `properties` parameter, it uses the Shadcn `useReactTable` hook with the columns and the data returned from the Gun JS database. This works in a similar way to the status table (See figure 5-33).


```

<div className='w-full'>
  <div className='mb-4'>
    {(data && data?.length > 0) && <Input
      placeholder='Filter by id'
      value={{(table.getColumn('id')?.getFilterValue() as string) ?? ''}}
      onChange={(event) =>
        table.getColumn('id')?.setFilterValue(event.target.value)
      }}
      className='max-w-sm mt-8'
    />
  </div>
</div>

```

Figure 5-34 Id filter bar for collection table.

At the top of the component, there is a filter bar for filtering the table by each entry's ID. This part of the component checks that data exists first and that its length is greater than zero. If this is true, a search bar is rendered. OnChange, there is a callback that takes in the event object and uses the table's ID column setFilterValue to update the table with the filtered result (See figure 5-34).

```

<TableBody>
  {table.getRowModel().rows.map((row) => (
    <TableRow key={row.id} className='border-none'>
      {row.getVisibleCells().map((cell) => {
        const isIdColumn = cell.column.id === 'id'

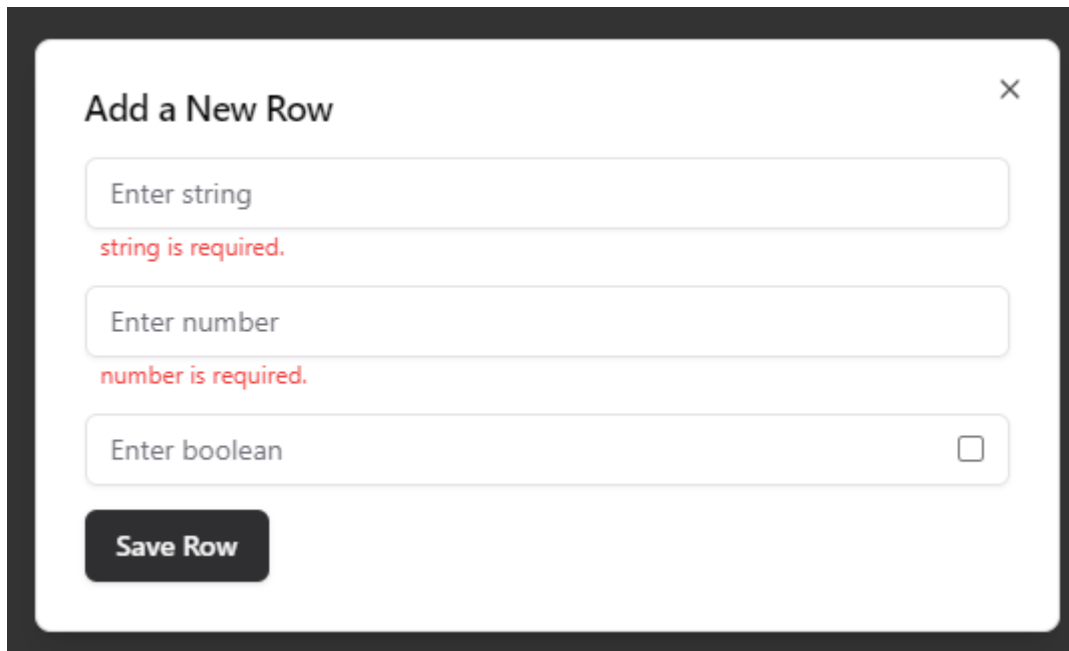
        return (
          <TableCell key={cell.id} className='border-none'>
            {isIdColumn ? (
              <div className='flex items-center'>
                <input
                  type='radio'
                  id={`radio-${row.id}`}
                  name='row-select'
                  onChange={() => {
                    setSelectedRow(row.original)
                    row.toggleSelected(true)
                  }}
                />
                <label htmlFor={`radio-${row.id}`} className='ml-2'>
                  {cell.renderValue() as string}
                </label>
              </div>
            ) : (
              cell.renderValue() as string
            )}
          </TableCell>
        )
      })}
    </TableRow>
  )}
</TableBody>

```

Figure 5-35 Table body component with radio for row select.

Additionally, in the table, there needed to be functionality to select a column. With a row selected, the user would be able to perform an update or delete on the item. This is done in the `TableBody` component. The rows are mapped, and a `TableRow` component is rendered. If the `TableRow` is the ID column, it will render a radio button. The radio button's `onChange` property sets the `selectedRow` to `row.original`, which contains the data added in the row (in this case, it is the ID string). If the `map` function is not mapping over the ID, it just renders the value as a string (See figure 5-35).

5.8.3 Item 2



The image shows a modal dialog titled "Add a New Row" with a close button in the top right corner. The dialog contains three input fields. The first field is labeled "Enter string" and has a red error message "string is required." below it. The second field is labeled "Enter number" and has a red error message "number is required." below it. The third field is labeled "Enter boolean" and has a checkbox to its right. At the bottom left of the dialog is a dark button labeled "Save Row".

Figure 5-36 Add new row component with validation.

The next item in this sprint was the inclusion of the add new row pop-up. Similar to the collection table, this component needed to dynamically generate the properties for the user-defined collection and have appropriate form validation for each field. This component was made using the `Shadcn Dialog` component, which works as a button that, when clicked, opens a pop-up component (See figure 5-36).

```

{properties.map((property, index) => {
  if (property.name === 'id') return

  if (property.type === 'boolean') {
    return (
      <div key={` ${property.name}-${index}`} className='mb-3'>
        <div className='flex items-center justify-between h-9 w-full rounded-md border px-3 py-1 text-sm shadow-sm'>
          <div className='text-[#71717A]'>Enter {property.name}</div>
          <input
            type='checkbox'
            checked={form[property.name] === 'true'}
            onChange={(e) => handleChange(property.name, e.target.checked ? 'true' : 'false')}
          />
        </div>
        <FormError message={errors[property.name]} />
      </div>
    )
  }
  else if (property.type === 'number') {
    return (<div key={` ${property.name}-${index}`} className='mb-3' >
      <Input
        type='number'
        value={form[property.name] as string || ''}
        onChange={(e) => handleChange(property.name, e.target.value)}
        placeholder={`Enter ${property.name}`}
      />
      <FormError message={errors[property.name]} />
    </div>)
  }
  else if (property.type === 'string') {
    return (<div key={` ${property.name}-${index}`} className='mb-3' >
      <Input
        type='text'
        value={form[property.name] as string || ''}
        onChange={(e) => handleChange(property.name, e.target.value)}
        placeholder={`Enter ${property.name}`}
      />
      <FormError message={errors[property.name]} />
    </div>)
  }
})}
<div className='flex gap-3'>
  <Button onClick={handleAddRow}>Save Row</Button>
</div>

```

Figure 5-36 Code for making dynamic add row form component.

This component dynamically creates inputs for each property based on what the collection model's properties have for the type parameter. If the type is a Boolean, a checkbox is rendered. If the type is a string, a text box is rendered, and if it's a number, a number box is rendered. Each input has a Form Error component that renders the error message if its validation fails. As this form is filled in, the form state variable is updated, which is then passed to the `validateForm` function before creating a new entry in the Gun JS database (See figure 5-36).

```

export const validateForm = (
  form: { [key: string]: string },
  properties: Property[],
  setErrors: Dispatch<SetStateAction<{ [key: string]: string }>>
) => {
  const formErrors: { [key: string]: string } = {}

  // by default keep has error as false then check if error exists
  let isValid = true

  properties.forEach((property) => {
    const value = form[property.name] || ''
    const error = validateFormField(property.name, value, properties)

    if (error) {
      formErrors[property.name] = error
      isValid = false
    }
  })

  setErrors(formErrors)
  return isValid
}

```

Figure 5-37 validateForm function.

```

const validateFormField = (name: string, value: string, properties: Property[]) => {
  const property = properties.find((prop) => prop.name === name)

  if (!property) return

  if (property.name === 'id') return

  if (value === '') {
    return `${property.name} is required.`
  }

  if (property.type === 'number' && isNaN(Number(value))) {
    return `${property.name} must be a valid number.`
  }

  if (property.type === 'string') {
    if (value.length < 3) {
      return `${property.name} must be at least 3 characters long.`
    }

    const regex = /^[a-zA-Z0-9 ]+$/
    if (!regex.test(value)) {
      return `${property.name} can only contain letters, numbers, and spaces.`
    }
  }
}

```

Figure 5-38 validateFormField helper function.

This function takes in a form, an array of properties, and an errors setter. The function works by using the type in the properties array and the value from the form and passing them to the `validateFormField` function, which is a switch statement with regex patterns for each type of property (string, Boolean, number). There is a catch-all conditional statement in the validation that checks that a value has been passed, and if the property passed to the function is the ID, it is skipped because the ID is generated by the app, not the user's input in the form, so it does not need to be validated. Numbers must be valid numbers, strings must have at least three characters, and they must not have any special characters. If any of the validation fails, the `validateForm` function returns false, preventing a new entry from being added to the database. Also, the errors are set from the validation to update the UI with the error message from the `validateFormField` function (See figure 5-37, 5-38).

5.8.4 Item 3

```
interface Props {
  selectedRow: Item
  setRefresh: Dispatch<SetStateAction<boolean>>
  model: Model
}

const UpdateRow = ({ selectedRow, setRefresh, model }: Props) => {
  const [form, setForm] = useState<{ [key: string]: string }>(selectedRow)
  const [open, setOpen] = useState(false)
  const [errors, setErrors] = useState<{ [key: string]: string }>({})

  useEffect(() => {
    setForm(selectedRow)

    if (!open) {
      setErrors({})
    }
  }, [open])

  const { properties } = model

  const handleChange = (name: string, value: string) => {
    setForm((prev) => ({ ...prev, [name]: value }))
  }
}
```

Figure 5-39 Update Row form component code.

The next item in this sprint was the inclusion of the update row pop-up button. Functionally, this works in almost the exact same way, but as you can see above, this button takes in a `selectedRow` parameter. The component renders the form like the `addRow` component, but with the `selectedRow` variable, the form is prefilled. This component uses the same code for validation, setting errors, and dynamically creating form inputs. This pop-up was also made using the `Shadcn Dialog` component. When the validation passes, instead of hitting the collection create route, it hits the collection update route with the new updated body (See figure 5-39).

5.8.5 Item 4

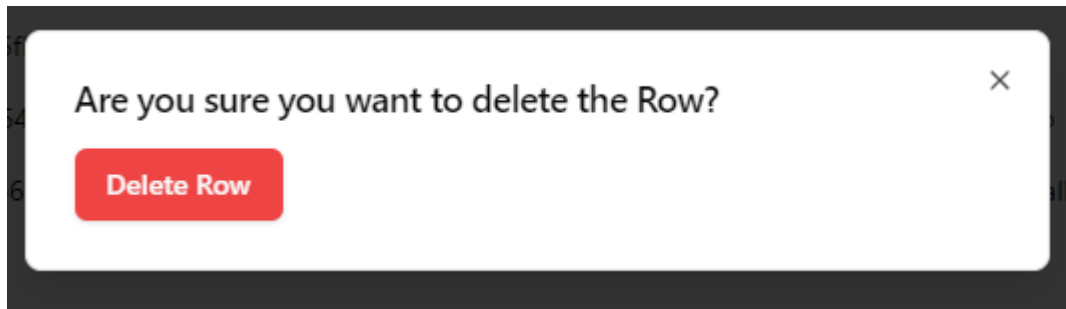


Figure 5-40 Delete row pop up component.

```
const handleDeleteRow = async () => {  
  const response = await deleteRow(modelId, selectedRowId)  
  
  if(response) {  
    resetPopUp()  
  }  
}
```

Figure 5-41 handleDeleteRow function.

Similarly, to create and update, delete collection was added to this sprint. This component was built using the same Shadcn Dialog component and the pattern of passing in a selected row to update the Gun JS database. The component pop-up has a button to verify that the user wants to delete the row. When clicked, the button calls `handleDeleteRow`, which is an asynchronous function that uses the `modelId` and the `selectedRowId` to hit the delete endpoint and delete the row. If a response is returned from the `deleteRow` function, the `resetPopUp` function is called, which closes the pop-up window (See figure 5-40, 5-41).

5.8.6 Item 5

```
export const POST = async (req: Request, { params }: { params: { singleId: string } }) => {

  const currentUrl = req.url

  try {
    const authHeader = await req.headers.get('Authorization')

    // verify token
    const checkToken = await authorisationMiddleware(authHeader)
    if (checkToken) return checkToken
    const { singleId } = await params

    const { value } = await req.json()

    if (value === undefined || value === null) {
      saveResponseStatus(currentUrl, 400)
      return NextResponse.json({ message: 'invalid params', ok: false }, { status: 400 })
    }

    const ref = gun.get('single')

    const body = { id: singleId, value: value }

    // generate an data integrity signature / encrypt data
    const signature = generateSignature(body)
    const encryptedData = encryptData(body)

    const newData = {
      encryptedData,
      signature,
    }

    ref.set(newData, (ack: Acknowledgment) => {
      if (ack.err) {
        console.error(ack.err)
        saveResponseStatus(currentUrl, 500)
        return NextResponse.json({ message: 'Failed to save data', ok: false }, { status: 500 })
      }
    })

    saveResponseStatus(currentUrl, 201)
    return NextResponse.json({ message: 'Data created', body: newData, ok: true }, { status: 201 })
  }
}
```

Figure 5-42 Singles create row route.

This item in the sprint was the inclusion of the singles page, which works very similarly to the collections page, but instead of the user-defined data having multiple properties, it is a single name-value pair. This took longer to implement than expected due to refactoring that had to be done in the API. From the UI side of things, creating rows and dynamic forms with validation was very similar to the collections, so adjusting the code and patterns was relatively easy. Previously, in the singles API, data was stored in a Gun reference under the singles' unique name. This had to be changed so that all singles are stored under the same ref, 'singles.' With this change, it reduces the complexity of the UI, as multiple API requests don't have to be made to get all singles. Now, one API request can be made, which accesses the singles ref in the Gun JS database and returns all the user-defined singles. Above is an example of these changes throughout the singles API. Now, the Gun reference comes from the string 'singles,' as opposed to the specific singles name. This was done for every route in the singles directory (See figure 5-42).

5.8.7 Item 6

```
export const GET = async (req: Request) => {

  const currentUrl = req.url

  try {
    const authHeader = await req.headers.get('Authorization')

    // verify token
    const checkToken = await authenticationMiddleware(authHeader)
    if (checkToken) return checkToken

    const ref = gun.get('users')
    const results: Item[] = []

    ref.map().once((res: EncryptedItem) => {
      if (res && res.encryptedData && res.signature) {
        const decryptedData = decryptData(res.encryptedData)
        const isValid = verifySignature(decryptedData, res.signature)

        // if the signature is valid it means the data has not been tampered with outside of the api
        if (isValid) {
          results.push(decryptedData)
        }
        else {
          // alert the user to the fact that unauth data has been altered / add in roll back feature
          console.error('unauth user altered data start rollback')
        }
      }
    })

    // this is my temp solution to this issue
    await new Promise(resolve => setTimeout(resolve, 1000))

    if (results.length === 0) {
      saveResponseStatus(currentUrl, 404)
      return NextResponse.json({ message: 'No users found', ok: true, }, { status: 404 })
    }

    saveResponseStatus(currentUrl, 200)
    return NextResponse.json({ message: 'Got all users successfully', ok: true, body: cleanResponse(results) }, { status: 200 })
  }
}
```

Figure 5-42 Get all registered user's route.

Additionally, in this sprint, the user view page was added, which shows what users have registered using the CMS. This simple route was added. After authenticating the user's API token, the route gets a reference to users in the Gun JS database. Then, using this ref, it applies the map and once methods on the ref. In the once callback, the data is decrypted, and its data signature is verified to ensure the data has not been tampered with. If the data is valid, it is pushed into the results array, which is then returned to the client at the end of the route. There is a check to ensure that users were found.

5.9 Sprint 5

5.9.1 Goal

This was a smaller sprint than previous ones. The primary goal for it was to write unit tests for the singles and collections API routes. Another item included in this sprint was the ignore header to prevent redundant API calls from being saved to the dashboard. The collection creator component had a bug that was fixed in this sprint.

5.9.2 Item 1

```
export const POST = async (req: Request, { params }: { params: { modelId: string } }) => {
  const currentUrl = req.url

  // this optional header is used to ignore saving the api response
  const ignoreHeader = req.headers.get('Ignore')
  const authHeader = req.headers.get('Authorization')

  try {
    // verify token
    const checkToken = authorisationMiddleware(authHeader)
    if (checkToken) return checkToken

    const { modelId } = await params
    const body = await req.json() as Item
    const ref = gun.get(modelId)

    if (!body || !modelId) {
      !ignoreHeader && saveResponseStatus(currentUrl, 400)
      return NextResponse.json({ message: 'invalid params', ok: false }, { status: 400 })
    }

    // generate an data integrity signiture / encrypt data
    const signiture = generateSigniture(body)
    const encryptedData = encryptData(body)

    const newData = {
      encryptedData,
      signiture,
      id: generateRowId()
    }

    ref.set(newData, (ack: Acknowledgment) => {
      if (ack.err) {
        console.error(ack.err)
        !ignoreHeader && saveResponseStatus(currentUrl, 500)
        return NextResponse.json({ message: 'Failed to save data', ok: false }, { status: 500 })
      }
    })
  }
}
```

Figure 5-43 Ignore header in collection create route.

The first item in this sprint was the inclusion of the ignore header. Whenever the user was navigating between pages on the dashboard, the CMS was saving all the individual API requests, which was making the application slower and adding redundant API requests to the status page. This pattern is now applied to every single route in the API. The route tries to access the ignoreHeader and assigns it to a variable. Whenever the saveResponseStatus function is called, there is a check to see if the ignoreHeader is not present using AND chaining. If the ignoreHeader is not present, the API will save the response, and it will show up on the status page. Otherwise, it will not be saved (See figure 5-43).

5.9.3 Item 2

```
import { addRowToCollection, deleteRow, getAllCollectionRows, updateCollectionRow } from '../../utils/api'

// this is used to mock the response
global.fetch = jest.fn()

const mockApiUrl = 'http://localhost:3000/'
const mockModelId = 'test-model-123'
const mockErrorResponse = { message: 'Error occurred' }

describe('getAllCollectionRows', () => {
  beforeEach(() => {
    // reset mock fetch
    (fetch as jest.Mock).mockClear()
    jest.spyOn(console, 'error').mockImplementation(() => {})
  })

  it('should make a successful GET request and return all rows for collection', async () => {
    // for some reason this line needs the semi colon
    const mockResponse = { body: [{ id: '1', name: 'Test Item' }] };

    // mock the response
    (fetch as jest.Mock).mockResolvedValue({
      ok: true,
      json: async () => mockResponse,
    })

    const response = await getAllCollectionRows(mockModelId)

    expect(fetch).toHaveBeenCalledWith(
      `${mockApiUrl}api/collections/${mockModelId}`,
      expect.objectContaining({
        method: 'GET',
        headers: expect.objectContaining({
          'Content-Type': 'application/json',
          'Authorization': `Bearer ${process.env.NEXT_PUBLIC_API_TOKEN}`,
          'Ignore': 'ignore',
        }),
      })
    )

    expect(response).toEqual(mockResponse.body)
  })
})
```

Figure 5-44 Sample jest unit test.

A big item in this sprint was the inclusion of unit tests for the API routes. Tests were written for all routes that require CRUD functionality. Jest has the functionality to mock responses, so you can correctly test the API's error handling and success responses without worrying about connections or database mutation. This also removes the need for a testing environment in this situation. Accessing the global fetch property and assigning it to jest.fn enables mocking responses. The API URL, model ID, and an error response are mocked globally, as they are used in other tests in this file. The first test suite, getAllCollectionRows, has a beforeEach callback that clears the previous mocked values. The first test creates a mock response and calls the getAllCollectionRows function using the mockModelId. The test expects the fetch to be called with the mockApiUrl and the mockModelId. It is also expected to have the JSON content type, authorization header, the ignore header, and be a GET request. The response is also expected to match the mock body. Tests matching this pattern of mocking responses and verifying they include the correct URL, and headers are seen throughout the collections and singles tests (See figure 5-44).

```
it('should handle non-OK response correctly', async () => {
  (fetch as jest.Mock).mockResolvedValue({
    ok: false,
    json: async () => ({ message: 'Error occurred' }),
  })

  const response = await getAllCollectionRows(mockModelId)

  expect(response).toBeUndefined()
  expect(console.error).toHaveBeenCalledWith('Network response was not ok')
})
)
```

Figure 5-45 Error handling test.

Additionally, there are tests to ensure the correct error messages are sent to the client if the network response is failing (See figure 5-45).

5.9.4 Item 3

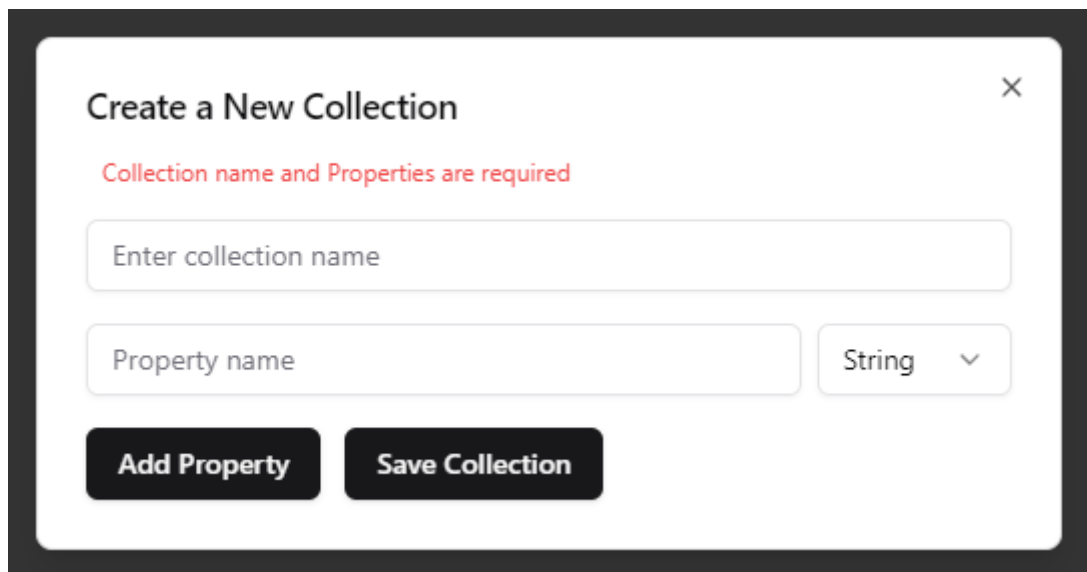
A screenshot of a web form titled "Create a New Collection" with a close button (X) in the top right corner. Below the title, a red error message reads "Collection name and Properties are required". The form contains two input fields: "Enter collection name" and "Property name". To the right of the "Property name" field is a dropdown menu currently showing "String" with a downward arrow. At the bottom of the form are two dark buttons: "Add Property" and "Save Collection".

Figure 5-46 Collection creator component with validation.

```

const validateNewCollection = (newCollection: Model) => {
  const { properties, name } = newCollection

  // if the properties length is one it means there is only the id and the collection
  is invalid
  if (properties.length === 1 || !name) {
    setError('Collection name and Properties are required')
    return false
  }

  setError(undefined)
  return true
}

const removePropertyField = (index: number) => {
  const updatedProperties = properties.filter((_, i) => i !== index)
  setProperties(updatedProperties)
}

const handleCreateCollection = () => {
  const newCollection = {
    modelId: `${newCollectionName}-${generateModelId()}`,
    name: newCollectionName,
    properties: [idTemplate, ...properties.filter((p) => p.name)],
    items: [],
  }

  if (validateNewCollection(newCollection)) {
    addNewModelToModels(newCollection)
    setNewCollectionName('')
    setProperties([{ name: '', type: 'string' }])
    setOpen(false)
    refreshPage()
  }
}

```

Figure 5-47 Collection creator validation code.

The create collection component had a bug where a user could define a model that had no properties. This was a significant bug that could break the UI and make endpoints in the API unreachable. The solution to this was the inclusion of the `validateNewCollection` function. This function takes in a model that the user is creating and checks that it has at least one property. If it does not, it sets the error message state variable and returns false. The `handleCreateCollection` function works by creating a `newCollection` variable with a unique `modelId`, `name`, `properties`, and an empty `items` array. If this `newCollection` passes the validation function, the new model is written to the public directory, and the component is reset (See figure 5-47, 5-46).

5.10 Sprint 6

5.10.1 Goal

The first item in this sprint was the development of an app separate from the CMS that uses it as a backend. Additionally, this item exposed a bug in the collections table, which was addressed in this sprint.

5.10.2 Item 1

Todos App

Add Todo

<input type="checkbox"/> test add from todos app	Delete
<input type="checkbox"/> new todo again	Delete
<input type="checkbox"/> show console	Delete
<input type="checkbox"/> new todo to do	Delete
<input checked="" type="checkbox"/> body	Delete
<input type="checkbox"/> new todo	Delete
<input checked="" type="checkbox"/> this actually is the new one	Delete

Figure 5-48 Sample Todo app which use the CMS as a backend.

The main item in this sprint was the development of a sample application that uses the decentralized CMS as a backend. This was done to highlight the CMS's capabilities while also finding bugs and testing to see what additional features are required. Originally, a note-taking application was going to be created, but this was limited in terms of features. To address this, a Todo tracker app was developed. The application was built using vanilla React and JavaScript, with Bootstrap styling for ease of development. In the application, users can create, read, update, and delete todos. They can also mark them as complete. The notes are all stored in the Gun JS decentralized database. Todos can be seen in the CMS and have CRUD functionality there as well, which updates the UI (See figure 5-48).

```

const [todo, setTodo] = useState({ id: '', body: '', isDone: false })
const [todos, setTodos] = useState([])

useEffect(() => {
  const handleGetTodos = async () => {
    const response = await getAllTodos()

    if (response && response.length > 0) {
      setTodos(response)
    }
  }

  handleGetTodos()
}, [])

const handleInputChange = (e) => {
  setTodo({ body: e.target.value, isDone: false })
}

const handleAddTodo = () => {
  createNewTodo(todo)
  setTodos([...todos, todo])
  setTodo({ body: '', isDone: false })
}

const handleToggleDone = (index) => {
  const updatedTodos = todos.map((todo, i) => {
    if (i === index) {
      const updatedTodo = { ...todo, isDone: !todo.isDone }
      updateTodo(updatedTodo)
      return updatedTodo
    }

    else {
      return todo
    }
  })
  setTodos(updatedTodos)
}

const handleDelete = (index, id) => {
  const updatedTodos = todos.filter((_, i) => i !== index)
  setTodos(updatedTodos)
  deleteTodo(id)
}

```

Figure 5-49 Todo app component code.

For each CRUD operation, a helper function was created in the todo app that simply makes a fetch request to the specified endpoint with the API key generated by the CMS. In the App route component's body, there are two state variables: `Todo` and `todos`. `Todo` represents the current todo that the user is creating, which by default has an `id`, a `body`, and an `isDone` property, the latter of which defaults to `false`. Below the starting state variables, there is a `useEffect` that makes an API request to the CMS to fetch all the todos. If the response is truthy and the length is greater than zero, it will call `setTodos`, which fills out the UI with all the user-created todos. There is a `handleInputChange` function that is called in the `onChange` property of the text box in the UI. This sets the current `Todo`, and when the user clicks the create todo button, the `handleAddTodo` function is called. This sends the new todo to the create route and adds it to the decentralized database. The function also pushes that todo to the `todos` array, so the application doesn't have to wait for the API to respond with the new todo added. There is also a `toggleDone` function. When a todo is displayed in the UI, the user can click the checkbox beside the todo to mark it as done. To handle this, the `handleToggleDone` function maps through all the `todos`, finds the one that matches the given index, then sets the todo's updated `isDone` property to the opposite of its current value (true or false). It

then sends the new data to the API to update the decentralized database. This also updates the todos array in the function, so the user doesn't have to wait for the `useEffect` to run and fetch everything again. There is also a `handleDelete` function that takes in a todo id and the index of the todo in the todos array. This function uses the `filter` method to remove the todo that matches the index passed into the function (the index of the selected todo to be deleted). It then updates the todos array with the filtered array and sends the todo id to the delete route to handle deleting the selected todo (See figure 5-49).

5.10.3 Item 2

```
export const transformBoolStringsInForm = (form: { [key: string]: string }) => {
  return Object.fromEntries(
    Object.entries(form).map(([key, value]) => {
      if (value === 'true') return [key, true]
      if (value === 'false') return [key, false]
      return [key, value]
    })
  )
}

// this does the opposite of the above function it takes an array of data and makes
// booleans
// a usable string for the ui
// chatgpt
export const transformBoolToStringValue = (data: Collection[]): Collection[] => {
  return data.map(item => {
    const transformedValues = Object.fromEntries(
      Object.entries(item).map(([key, value]) =>
        typeof value === "boolean" ? [key, value.toString()] : [key, value]
      )
    )
    return { ...item, ...transformedValues }
  })
}
```

Figure 5-50 `transformBoolStringsInForm` and `transformBoolToStringValue` functions.

During the process of creating this application, a major bug was found in the CMS. When creating a `Todo`, the `isDone` property was being sent to the API as a Boolean value, but in the CMS, it was adding them to the database as a stringified `'true'` or `'false'`, as opposed to an actual Boolean value. This caused two large issues. The stringified Booleans from the CMS caused the `Todo` app to not render correctly. Also, the actual Boolean values did not render on the CMS because the CMS was expecting a string, not a Boolean.

To solve this bug, two functions were written: `transformBoolStringsInForm` and `transformBoolToStringValue`. The first function takes in a form object, which comes from the `add to collection` and `update collection` components. Then, mapping the object entries, the callback checks if the values are stringified `'true'` or `'false'`, and then converts them to the actual Boolean values. This is done before adding Boolean values into the *Gun JS* database.

Another function had to be written to do the opposite of this, so true Boolean values could be displayed in the database. This was harder to implement than initially expected, so ChatGPT was used to write this function. The function takes in a collection array and maps through it. In the callback, it maps each collection in the array and converts the true Boolean values to their stringified counterparts. This is used in the collections table when a Boolean value needs to be displayed in the table (See figure 5-50).

5.11 Sprint 7

5.11.1 Goal

The primary goals for this sprint were to develop the last features (configure peers and rich text) in the CMS, and to develop another sample application that shows off the features of the CMS.

5.11.2 Item 1

```
return (
  <Dialog open={open} onOpenChange={setOpen}>
    <DialogTrigger asChild>
      <Button className='ml-3 mt-4'>Add New Peer</Button>
    </DialogTrigger>
    <DialogContent>
      <DialogHeader>
        <DialogTitle>Add a New Peer</DialogTitle>
      </DialogHeader>
      <div>
        <div className='mb-3'>
          <Input
            type='text'
            value={url}
            onChange={(e) => handleChange(e.target.value)}
            placeholder='Enter peer URL'
          />
        </div>
        <div className='flex gap-3'>
          <Button onClick={handleAddPeer}>Add Peer</Button>
        </div>
      </div>
    </DialogContent>
  </Dialog>
)
```

Figure 5-51 Configure peer component

The first item in this sprint was the inclusion of the configure peer's component. This will enable the developer using the application to configure peers from the UI, as opposed to editing the env file manually. By default, the CMS is configured with the *Gun JS* public network's open peer, which is

written into the env file on project setup. Additional peers are stored in a peers.json file in the public directory. A route was created to add a peer URL to this JSON file. Whenever the *Gun JS* library is used, this JSON file is read, and the Gun object is instantiated with these peers. A basic configure peer component was made, which allows the user to add new peer URLs to this JSON file by hitting the endpoint (See figure 5-51).

5.11.3 Item 2

```
const editor = useEditor({
  content,
  onUpdate: ({ editor }) => {
    const content = editor.getHTML()
    handleChange(name, parseRichText(content))
  },
  extensions: [
    StarterKit,
    Placeholder.configure({
      placeholder: `Enter ${name}`,
      emptyNodeClass:
        'first:before:text-gray-400 first:before:float-left first:before:content-[attr(data-placeholder)] first:before:pointer-events-none',
    }),
    Heading.configure({
      HTMLAttributes: {
        class: 'text-xl font-bold capitalize',
        levels: [2],
      },
    }),
    ListItem,
    BulletList.configure({
      HTMLAttributes: {
        class: 'list-disc ml-2',
      },
    }),
    OrderedList.configure({
      HTMLAttributes: {
        class: 'list-decimal ml-2',
      },
    }),
  ],
  immediatelyRender: false,
  editorProps: {
    attributes: {
      class:
        'shadow appearance-none min-h-[150px] border rounded w-full py-2 px-3 bg-white text-black text-sm mt-0 md:mt-3 leading-tight focus:outline-none focus:shadow-outline',
    },
  },
})
```

Figure 5-52 Richtext config object.

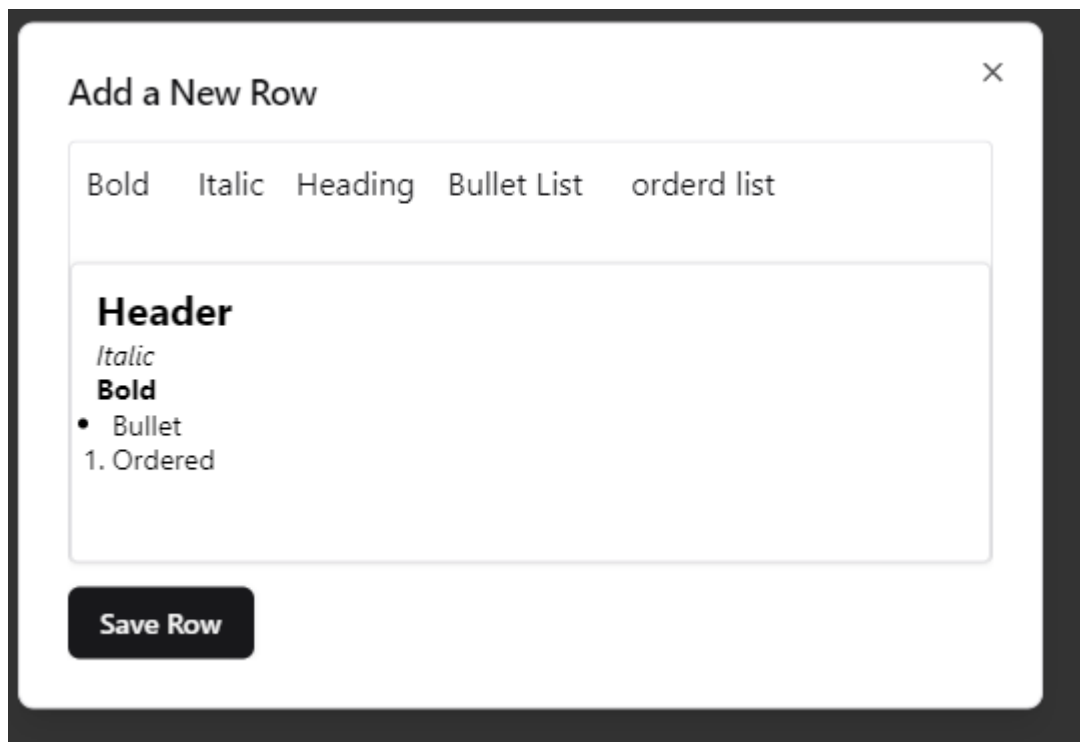


Figure 5-53 Richtext component in create row.

The next item in this sprint was the inclusion of the rich text property to the CMS. Before adding this, the CMS had the capability to have Booleans, numbers, and plain string values. Adding in a rich text editor introduced a new level of complexity beyond a plain string. A rich text editor is a body of text that has built-in markup capabilities. For the rich text editor in this application, the user could write a paragraph and have the text be a header style, bold, italicised, numbered list, or a bullet list.

To ease the development of this feature, the Tip Tap framework was used. Tip Tap is a React library that makes creating rich text editors much easier. It does this by using built-in components and configs for generating them. The developer just must configure the config correctly with its functionality and styling, then pass it into the EditorContent component (See figure 5-52). The config is set using the useEditor Tip Tap hook. There is an optional variable, content, sent to the config — this is for when users are updating data. There is an onUpdate property, which sets a state variable in the parent with the rich text value, which is plain HTML that is sanitized before being added to the DB to prevent bad actors from sending malicious content to the database. The rest of the config consists of CSS styles for the heading and list items (See figure 5-53).

5.11.4 Item 3

```
export const encryptData = (data: Item | User) => {  
  const json = JSON.stringify(data)  
  
  // Generate AES key + IV  
  const aesKey = crypto.randomBytes(32) // AES-256  
  const iv = crypto.randomBytes(16)    // AES IV  
  
  // Encrypt data using AES-256-CBC  
  const cipher = crypto.createCipheriv('aes-256-cbc', aesKey, iv)  
  let encryptedData = cipher.update(json, 'utf8', 'base64')  
  encryptedData += cipher.final('base64')  
  
  // Encrypt AES key using RSA  
  const encryptedKey = crypto.publicEncrypt(  
    {  
      key: process.env.PUBLIC_RSA_KEY as string,  
      padding: crypto.constants.RSA_PKCS1_OAEP_PADDING  
    },  
    aesKey  
  ).toString('base64')  
  
  return {  
    encryptedData,  
    encryptedKey,  
    iv: iv.toString('base64')  
  }  
}
```

Figure 5-54 updated encryptData function.

During the user testing and creation of the rich text editor component, a major limitation of the CMS was found. Due to the use of RSA encryption, which is limited to 256 bytes for data it can safely encrypt, this was fine before the rich text component was added. But with its inclusion, the size of data being added to the database was much larger, and this would have to be improved.

The solution to this size problem was to use AES encryption along with SHA encryption, which is a common convention in software development. The encrypt and decryptData functions were updated using ChatGPT to reflect this change in design. In figure 5-69, the improved encryptData function can be seen. This function takes in the JSON to be encrypted, then generates a random aesKey along with an iv key. iv stands for initialization vector and is random noise added to the encryption so that even if the same data is encrypted, the output will be different. Using the AES key and the iv, the JSON is encrypted. Then the AES encryption string is encrypted again using the public RSA key. The twice-encrypted data, along with the AES decryption key and the iv, are returned in the function.

The encrypted data needs to be decrypted with the private RSA key to reveal the AES encryption string, which then can be decrypted using the encrypted key in the return statement. AES does not have a size limit when encrypting data, so this solves the "too large payload" error with RSA (See figure 5-54).

5.12 Sprint 8

5.12.1 Goal

This sprint had one primary goal, which was to find and solve as many bugs as possible in the application. This was done to improve the overall usability and performance of the CMS.

5.12.2 Item 1

Bug	Recreate	Status
Built application cannot write to public directory	Run npm run build try create a new collection	Done
Unable to login on login route	In rest client or hospital app hit /api/UserAuth/login with email and password body	Done
Singles not pushing to table when changed	perform crud on single page	In progress
No rows found before rows are filled from db (improve loading logic)	Click collection that has rows already added to db in the inner side bar	Todo
Stop stringifying	add number	

Figure 5-54 Bug spreadsheet for tracking progress.

As mentioned in 5.12.1, the purpose of this sprint was to find and solve bugs and overall make improvements to the code of the application. The process of doing this was carried out by building

the CMS and running the test applications that use it, then trying to find bugs by completing tasks. The bugs were tracked in a spreadsheet with a description of the issues and steps to recreate it, along with its status. This ensured the bugs would be managed and solved within the sprint. Whenever a new bug was found, the spreadsheet would be updated (see figure 5.54).

5.12.3 Item 2

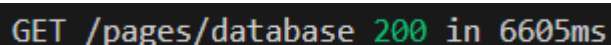
```
export default async function Status() {  
  
  const statusArray = await getResponseStatus()  
  
  return (  
    <div className='flex'>  
      <Sidebar />  
      <div className='p-4 ml-20'>  
        <Title firstPartOfTitle='API' secondPartOfTitle='Logs' />  
        <StatusWrapper statusArray={statusArray} />  
      </div>  
    </div>  
  )  
}
```

Figure 5-55 Status page converted to an asynchronous server component

At this point in the process, the main features of the CMS were developed, but there was a major issue with the CMS: the performance of the application was very slow. Some pages would take upwards of ten seconds to load in. There were two primary reasons for the poor performance of the CMS. The first was the size and complexity of the project—the project was very bloated and needed optimisation.

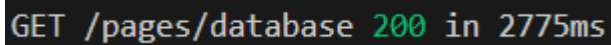
Additionally, Next JS can prerender components on the server and even make API requests, then serve the client the component with the API data. None of these features were used before this item in the sprint. To improve the performance, each page in the CMS was converted into a server component that would get the data, then serve the client with the data already preloaded. This reduced the load time significantly.

This can be seen in the status page. The page is loaded asynchronously, which can be seen at the page function declaration. The page makes an await request to the `getResponseStatus` function, which returns the API status data. This is then passed to the `StatusWrapper` component, which is the bundled client component for this page (See figure 5-55).



```
GET /pages/database 200 in 6605ms
```

Figure 5-56 Database load time before server components



GET /pages/database 200 in 2775ms

Figure 5-57 Database load time after server components

The load time was almost halved when loading the database page with and without server components (See figure 5-56, 5-57).

5.12.4 Item 3

```
const DB_NAME = `indexDB-${process.env.NEXT_PUBLIC_IDB_STORE}`
const STORE_NAME = `models-${process.env.NEXT_PUBLIC_IDB_STORE}`

export const getDb = async () => {
  return await openDB(DB_NAME, 1, {
    upgrade(db) {
      if (!db.objectStoreNames.contains(STORE_NAME)) {
        db.createObjectStore(STORE_NAME, { keyPath: 'modelId' })
      }
    },
  })
}

export const saveModelToIndexedDB = async (model: Model) => {
  const db = await getDb()
  await db.put(STORE_NAME, model)
}

export const getAllModelsFromIndexedDB = async (): Promise<Model[]> => {
  const db = await getDb()
  return await db.getAll(STORE_NAME)
}
```

Figure 5-58 IndexedDB utility file.

Another optimisation and bug fix came from removing the dependence on JSON files in the public directory. The application wrote and read from the public directory for peers, API status response handling, first-time login, and managing models. When running in development, this was optimal as it was very fast and allowed shared data between the client and the server. Unfortunately, when the application is built, files in the public directory are treated as static and cannot be mutated. This caused many issues in the application; for example, new models could not be created.

To solve this problem for the models, IndexedDB was used. IndexedDB is a built-in database in every browser that stores key-pair values that can persist between refreshes and cache clears. Each CMS project generates its own IndexedDB store and stores the models there. This enables new models to be created in a built project, and IndexedDB performance is better than reading a JSON from the server. This also enables multiple CMS to run on the same client without unnecessarily sharing models.

To ease the use of IndexedDB, the IDB library was used. This removes some of the boilerplate from writing to IndexedDB in vanilla TypeScript. Three functions were made for saving models, retrieving models, and generating the IndexedDB store. Along with IndexedDB, models are also stored in Gun JS as a fallback, so if the user accesses the CMS from a different client, they will still be able to use their models even if they are not cached in IndexedDB (See figure 5-58).

5.12.5 Item 4

```
export const validateOnServer = (body: Item, properties: Property[]) => {

  const errors: { [key: string]: string } = {}

  if(!bodyPropertiesMatch(body, properties)) {
    errors['invalidBody'] = 'Body sent to API does not match model'
  }

  for (const property of properties) {
    const { name, type } = property
    const value = body[name]

    if (name === 'id') continue

    if(!value) continue

    // rich text needs special validation as it is stored as a string with special
    // characters
    // not a specific richtext type
    if (type === 'richtext' && value[0] !== '<') {
      errors[name] = `Invalid ${name} type`
    }

    if(type !== 'richtext' && typeof value !== type) {
      errors[name] = `Invalid ${name} type`
    }

    if(type === 'string') {
      if (value.length < 3) {
        errors[name] = `${name} must be at least 3 characters long.`
      }

      const regex = /^[a-zA-Z0-9 ]+$/
      if (!regex.test(value)) {
        errors[name] = `${name} can only contain letters, numbers, and spaces.`
      }
    }
  }
}
```

Figure 5-59 Server validation.

Another improvement made to the CMS was in the server. Up to this point, the only validation was done on the actual CMS client; no server validation was done in the resource routes. This is needed to ensure the security and safety of the API. The function takes in the body being added to the

database and the resources model. It then compares them to ensure the body does not contain properties not specified in the model. It's worth noting that the body does not need to contain every property in the model; for example, if the user is updating only one property on a row in the database, this validation will still work. Rich text types must start with a greater-than symbol to be valid rich text. For numbers and strings, the types must match the model using the `typeof` operator. Ordinary strings must not contain special characters and must be at least three characters long (See figure 5-59).

5.12.6 Item 5

```
export async function middleware(request: NextRequest) {
  const isAuthenticated = request.cookies.get('authenticated')?.value === 'true'

  if (!isAuthenticated) {
    return NextResponse.redirect(new URL('/', request.url))
  }

  return NextResponse.next()
}

export const config = {
  matcher: '/pages/:path*',
}
```

Figure 5-60 Improved authentication middleware.

The final task completed in this sprint was improving the MetaMask authentication to something more robust, which improves performance. The main body of this task was moving the current user authentication from a custom hook and context to a middleware function and a cookie. The outdated version would get the account variable when the user authenticated using the MetaMask wallet, then write it to a custom React context. The custom React hook for authentication would then check this context; if the user was not authenticated, they would be redirected to the login page. This custom hook used a `useEffect`, which would cause multiple redirects and re-renders, greatly slowing down the performance when navigating between pages. The new version takes the same account variable and writes it to a cookie, which is then checked in middleware that runs when the user goes to a page route. The logic is the same, where it checks the cookie and if it's not present, it will redirect to the login route.

5.13 Conclusion

In summary, this chapter covered the specific development process involved in building this application. The application enables users to create a decentralized back-end for an external application in a user-friendly way, greatly reducing the complexity when developing decentralized systems. Users can create custom collections with properties for four different types: Boolean, string, numbers and rich text. Users can perform full CRUD on these either from the CMS dashboard or from the API generated from the project. The CMS also includes single valued routes. These can manage single values in an application or feature flags in external applications. The CMS also provides decentralized user authentication from the API. Additionally, the CMS dynamically generates documentation based on the resources the user created.

The purpose of this section was to document the development process of the decentralized CMS. This chapter demonstrates the experience gained with Next JS, Gun JS, IndexedDB and MetaMask SDK while also displaying project management and problem-solving skills. Finally, this section shows projects focus on security and optimisation.

Chapter 6. Testing

6.1 Introduction

This section dedicated to the testing of the decentralized CMS. The purpose of testing in the application is to find functional bugs in the implementation of the application and improve the overall usability. This section is split into two primary focus areas.

1. Functional testing
2. User testing

Functional testing is the process of verifying that the systems feature work as expected. This includes the testing of API routes, CRUD operations and navigation between pages in the CMS. The primary purpose of this is to find bugs within the application.

User testing focuses on the usability of the application. This process involves getting real users to complete tasks within the application. Design and accessibility issues were identified during this process. The users provide feedback for improving the application.

6.2 Functional Testing

This section covers the functional testing done on the CMS. This was done using the black box testing method. This is where the internal logic is not relevant to the tester, only that the expected outcome is met. These functional tests fall under three categories.

1. Navigation
2. API
3. UI CRUD

The navigation handles navigating between pages in the CMS for authenticated and un-authenticated users. The API testing is specifically for testing the API CRUD routes in a rest client not on the CMS UI. UI CRUD is the opposite of this, it tests performing CRUD from within the application user interface itself.

6.2.1 Navigation

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
1	Authenticate and redirect to database page	Click the login with MetaMask account button / type in password	MetaMask extension will pop up and the user can authenticate and redirect to the database page	MetaMask extension will pop up and the user can authenticate and redirect to the database page	Passed

2	Use side nav to navigate to the singles, status, profile, user's pages	Click on the tabs in the side nav	App redirects to selected page	App redirects to selected page	Passed
3	Un authenticated users should redirect to login page when trying to access database page	Unauthenticated user changes the URL to /database	Redirects to route login page	Redirects to route login page	Passed

6.2.2 API

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
1	Add new row to existing collection	Send a body object with collection to collection create route	Return status 201 with id, encrypted data and data signature	Return status 201 with id, encrypted data and data signature	Passed
2	Get all rows in a collection	Send a GET request to the API with model id query param	Return all rows in the specified collection decrypted	Return all rows in the specified collection decrypted	Passed
3	Get single row in a collection	Send a GET request to the API with model id and row Id	Return single selected row	Return single selected row	Passed
4	Update selected row	Send PUT request to API with model id and row	Return status 200 with id encrypted data and data signature	Return status 200 with id encrypted data and data signature	Passed

		id query params			
5	Delete selected row	Send DELETE request to API with model id and row id query params	Return 200 status with success message	Return 200 status with success message	Passed
6	Get all rows in a collection without an auth API token	Send a GET request to the API with model id query param, exclude the auth token header	Return status 401 with error message	Return status 401 with error message	Passed

6.2.3 UI CRUD

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
1	Create new model with collection creator component	Open collection creator component put in three properties for bool string and number	New collection added to side bar	New collection added to side bar	Pass
2	Add new row to collection	Open add new row component add in values for three properties	New row added to the table	New row added to the table	Pass

3	Update row	Select row and click update, make change to every property	Row is updated in the table	Row is updated in the table	Pass
4	Delete row	Select row and click delete button	Row is removed from table UI	Row is removed from table UI	Pass

6.2.4 Discussion of Functional Testing Results

The functional testing in this section evaluates and validates the functionality of the core features of the CMS.

Navigation:

Test 1: This test was a success, using MetaMask the user authenticated then redirected to the database page. This verifies the authentication middleware is functional and met the expected outcome.

Test 2: This test was a success. The user could navigate between the pages using the side navigation bar and the loading logic was successful rendering the correct page.

Test 3: This was a success. If an unauthenticated user altered the URL to redirect to a page the authentication middleware would redirect them to the login page. This verifies the security of the routes in the CMS.

API:

Test 1: This test ran successfully. There were no issues when creating a new row and adding it to the decentralized database returning a success message to the client.

Test 2: This ran correctly all data for a selected collection was returned and decrypted from the database, this verifies the decryption is working along with the GET all route.

Test 3: This test met the expected outcome, the selected row from the database was decrypted and returned to the client.

Test 4: This test ran successfully, the selected row was updated, and the success status was returned to the client.

Test 5: This was a successful test; the user deleted the selected row from the database returning a successful response to the client.

Test 6: This test was a success, the API rejected the request for data because the authentication header was not present, returning an unauthorized response.

UI CRUD:

Test 1: This was a successful test; the user created the model with three properties.

Test 2: This test matched the expected outcome; the user selected the created model and added a new row to the database.

Test 3: This test was a success, The user selected the created row and made an update request, the data was modified with a success response status.

Test 4: The delete functionally worked as expected, the user selected the row to be deleted and pressed the delete button.

6.3 User Testing

The purpose of the user testing is to evaluate the usability and user experience of the CMS. Additionally, it helps identify bugs or usability issues with the application. The user tests were conducted on three software developers as they are the target audience for the CMS. They were assigned three tasks to complete – create a model, add a row to the database and update the row. When they were completing these tasks their interactions with the user interface were recorded to ensure there were no usability issues. In addition, the developers were timed while completing the tasks. After the tasks were complete feedback was gathered from the users. The tasks were designed to replicate typical CMS usage, and the primary focus of the testing was on the database interactions.

User Id	Create model task time	Add row task time	Update row task time	Thoughts on create task	Thoughts on add task	Thoughts on update task
1	9 seconds	16 seconds	31 seconds	"Straight forward easy to add properties"	"Same as create very simple to add rows"	"Had issues finding the update button"
2	8 seconds	10 seconds	36 seconds	"No problems defining a model"	"Very simple and functional"	"Wasn't clear on how to update"
3	12 seconds	17 seconds	23 seconds	"Quick and easy to find and create"	"The data in the form was in a different order to the model I created"	"No problems updating the data"

6.3.1 Discussion of the user testing

All three software developer testers were able to complete the tasks in a reasonable time. During the feedback, the developers expressed issues with update functionality. The update button is not visible until the user selects a row from the database table. Additionally, a small issue was found where the add row form properties were in a different order to the model create and the database row. This caused minor confusion for the tester.

Despite these issues, the testers had positive feedback saying they would all use this product again in the future. The testers suggested improvements based on the issues they had while using the CMS these include making it clearer how to update and delete rows. In future iterations these usability issues will be fixed, improving the overall user experience.

6.4 Conclusion

In conclusion this chapter discussed the testing process for the decentralized CMS. The results for the functional testing performed as expected without any key issues or bugs. The user testing revealed minor issues with the usability of the CMS that in future iterations will be improved on, to provide a better user experience.

Chapter 7. Project Management

7.1 Introduction

The purpose of this section is to describe the management of the software project from start to finish. Project management and organization were paramount in ensuring the project requirements and goals were met. This chapter discusses every phase in the process, covering how the project evolved and why proper management was essential for a successful application.

Several tools were used for project management. The primary tool used was Trello; this organized the project backlog, which tracked progress on tasks to be completed during each phase of the project. Additionally, a combination of a `todo.txt` file and a personal journal was used specifically for breaking down larger tasks into smaller tasks, along with reflections and ideas. GitHub was the tool used for version control of the CMS. This ensured the project source code was managed correctly. This section concludes with a critical reflection on the project management process, discussing learning experiences and challenges faced throughout the project.

7.2 Project Phases

The project was divided into defined phases for a structured approach to the creation of the CMS. Each phase had a specific focus. This approach ensured that specific goals could be established and met for each distinct phase. Additionally, having phases helped maintain focus throughout the entire project while supporting continuous feedback from the project supervisor.

7.2.1 Proposal

The project started with the proposal phase, establishing a foundation for the rest of the project. The goal of this phase was to establish the focus of the project and identify research topics. The aim for the project was chosen during this phase: to create a decentralized content management system which would ease the development of decentralized applications. The research section for this project was written during this phase, focusing on types of decentralized storage and content management systems. The integration of these two technologies was also explored, highlighting their potential benefits.

Additionally, during this phase, the project scope was defined, including planned features and project goals. A formal proposal document was written. Following this, a project proposal meeting was held with the supervisor to discuss the project overview and receive feedback. This phase was vital for establishing the project goals and setting realistic deliverables.

7.2.2 Requirements

The requirements phase was critical to the development of the project. It created clear and achievable goals based on user requirements, as opposed to assumptions. This phase consisted of researching the technical feasibility and user needs; subsequently, this would create the foundation for the design and implementation phases. A feature questionnaire for the target audience and a competitor analysis established the functional requirements.

The technical feasibility was explored during this phase. A major challenge was the limited decentralized frameworks with low-quality documentation. To overcome these challenges, demo

applications were developed to explore these tools. Ultimately, Gun.js was selected for the decentralized database due to its simplicity compared to other options such as the IPFS protocol.

Overall, the requirements were managed successfully. Supervisor meetings were essential for estimating project scope and providing guidance during the technical feasibility section.

7.2.3 Design

The design phase played a critical role in the development of the decentralized CMS. Figma was used for both the software architecture and the user interface; this ensured alignment between both the functional and technical requirements. A major challenge encountered was designing the architecture. Decentralized systems introduce an elevated level of complexity, especially when incorporating safe user authentication and encryption. Supervisor feedback and support were essential for navigating these challenges. During the supervisor meetings, discussions were held about the project design, along with proposed frameworks for development. This guidance was particularly important for the CMS architecture.

7.2.4 Implementation

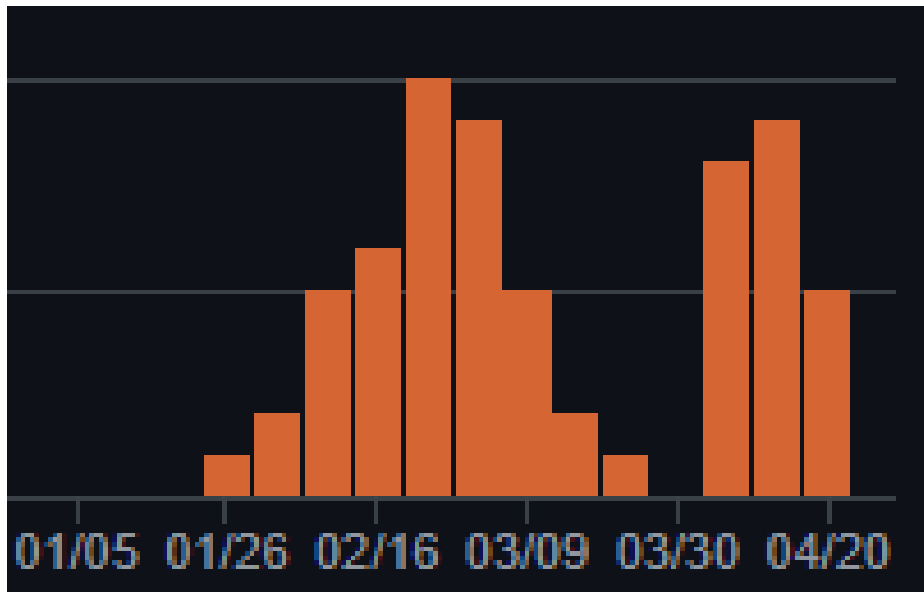


Figure 7-1 GitHub commits on project repository

The entire project followed the Agile framework; this was especially important during the implementation phase. Managing large requirements by breaking them into small tasks and tracking them in a Trello backlog ensured steady progress and planned execution.

Next JS simplified the process significantly by following the monolith design pattern. By combining both backend and frontend, the development process was managed more efficiently. Additionally, by abstracting complicated decentralized user authentication, the MetaMask SDK was essential to the implementation phase.

A significant challenge during this phase was over-scoping certain tasks. Tasks were selected for each sprint based on an estimated level of complexity. Sprint number three was an example of this (see

Chapter 5.7). Sprint five had too many tasks whose complexity was underestimated. This created a large workload, which caused difficulties, specifically with technical debt and poor optimization. Eventually, these problems were fixed in future sprints, but better planning would have prevented these issues. The workload completed in this sprint can be seen above in the repository insights commits graph (see Figure 7-1).

Additionally, adaptability was essential during this phase of development. Proper planning prevented issues, but not everything could be predicted. Unexpected bugs and issues came up, which caused changes to the development process. An example of this was the inclusion of AES encryption due to the size limitations of RSA.

Overall, this phase was managed successfully. In future developments, better planning will be implemented to prevent over-scoping, leading to a better-structured and more manageable development experience. This phase highlights the importance of planning and adaptability.

7.2.5 Testing

The Jest testing framework provided automated API testing, which made the development process more streamlined. Before committing changes, the tests would be run to ensure the API was not affected. This made the API development process more manageable. Additionally, two sample React web applications that use the CMS as a backend were developed to test the CMS functionality. This revealed logical errors in the code and enabled the expansion of the CMS.

User testing was conducted with three real users. These users were experienced software developers who were given tasks to complete—create a model and add and update a row in the database. The users were timed while completing these tasks and interviewed once finished. Their interactions were recorded in the personal journal. Having actual users provide feedback on the application was essential, and their suggested improvements will be addressed in the future.

7.3 SCRUM Methodology

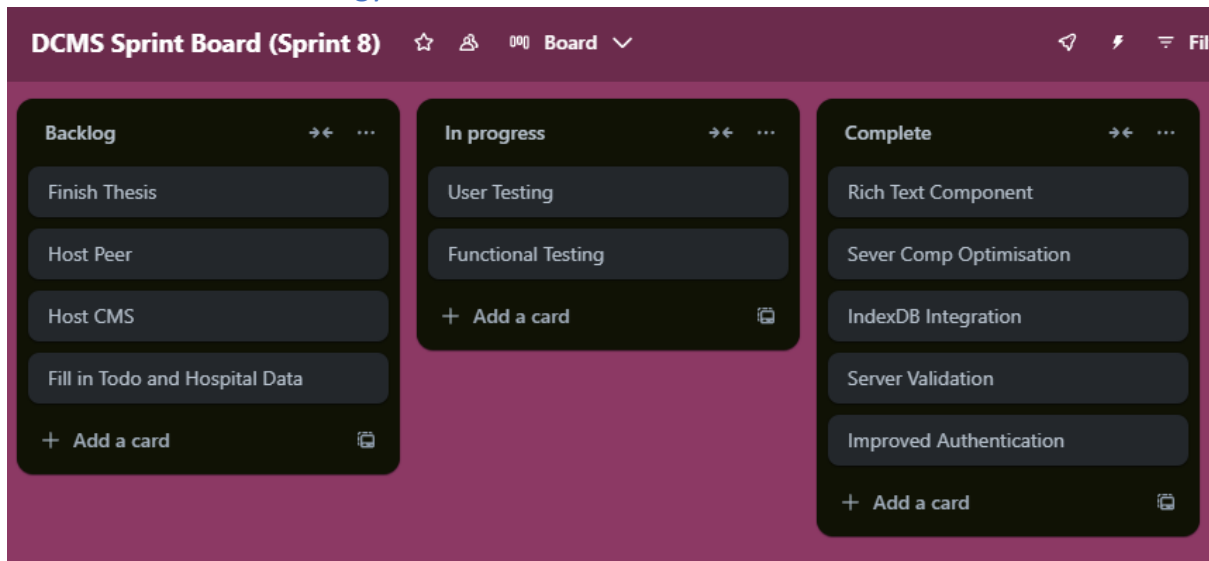


Figure 72 Trello board from sprints.

The project was divided into eight two-week sprints. Tasks were stored in a backlog in the Trello application; this enabled proper planning for each sprint. A story point value, which is an estimation of the level of difficulty a specific task has, was assigned to each task in the backlog. At the start of each sprint, tasks were moved from the backlog lane to the in-progress bar (see Figure 7-2).

The tasks were generated based on the research and requirements phases. Most of the tasks were specific coding features that were to be implemented. The SCRUM methodology ensured that each sprint was organized and planned, improving the development of the CMS.

7.4 Project Management Tools

7.4.1 GitHub

GitHub and Git were used for version control in this project. This ensured the project was managed correctly and enabled changes to be made to the repository from different clients. An extremely simplistic workflow was established for making changes to the project. Every change would be pushed to the main branch with a commit message that matched a selected task from the Trello board.

7.4.2 Journal

Throughout the entire process, a personal journal was kept. This was used to prototype ideas or take notes during supervisor meetings. Having this informal journal benefited the project; being able to quickly brainstorm ideas and solve problems was essential in developing the CMS.

7.5 Reflection

7.5.1 Views on the project

Ultimately, the project met the required results in developing a decentralized content management system, making this a major success. Reflecting on the experience, overcoming challenges, designing a complicated system and producing a polished final application has been incredibly rewarding. The experience gained in managing a large software project has been extremely valuable. Learning new paradigms and approaches for secure decentralized software was extremely challenging. Despite this, overcoming these obstacles not only contributed to the success of this project, but also provided invaluable experience and knowledge gain in the security and decentralized technology fields.

An important learning outcome from this project was the importance of strategy and planning. The requirements and design phases ensured proper planning for the success of the implementation section. Additionally, the use of the agile framework ensured consistent structured work while enabling constant feedback and improvement.

7.5.2 Working with a supervisor

Working with Mohammad was invaluable to the development and completion of this project. His experience and guidance were essential, without this specially in the planning and requirements gathering phase the project would not have been feasible. Additionally, Mohammad helped in expanding and formulating the proposal and recommending frameworks – Next JS, Shadcn which greatly improved the development experience. Ultimately, creating a decentralized content management system would not have been possible without the supervisor support and feedback.

7.5.3 Technical skills

A major benefit gained from the large software project was improvements in problem solving along with technical planning and learning new frameworks. These technical skills are essential in software development. The biggest challenge in relation to this was learning Gun JS and learning the fundamental paradigms of decentralized technology. While Gun JS abstracts a lot of complexity it is still very different to traditional database solutions. Additionally, securing decentralized applications is extremely complicated. While these were challenging it was incredibly rewarding developing these skills, improving networking and security skills which are essential to secure software.

While Next JS improved the developer experience greatly it came at a reasonably high learning curve, specifically with the API development and edge-based middleware. In addition to learning more about these, front end development skills were improved significantly. Specifically in relation to Typescript and optimising a large, structured project. Using Typescript for this project came with challenges, since users could define their own models that could have an arbitrary number of properties with several different types figuring out how to handle that in a strongly typed language was challenging. Solving difficult problems like this improved programming and problem-solving skills.

7.6 Conclusion

As mentioned above this project met required result of developing a decentralized content management system making is a significant achievement and major success. During this process many complex challenges were overcome. Examples of these are learning new decentralized paradigms, solving security and performance problems, and designing software architecture. These are essential skills which provide a strong foundation for a future in decentralized software development.

The experience gained from managing a large-scale project was incredibly educational, improving management and critical thinking skills. Additionally, working with a supervisor was invaluable to the process, providing feedback and guidance ensured the project was done to a high standard while developing essential skills for a future in software development. Additionally receiving advice on what tools and frameworks to use along with discussion on the feasibility of the decentralized CMS was essential to this project's success. Additionally, the project provided the opportunity to develop technical skills specifically in Next JS, Gun JS and Typescript.

Chapter 8. Conclusion

The primary focus of the application was the integration of decentralized storage technologies with content management systems to solve common issues with traditional centralized CMS solutions. An example of this is single points of failure and a higher risk of data loss. The primary goal of this project was to develop a user-friendly decentralized CMS. This was achieved by making it significantly easier and more efficient to create decentralized applications. Security was a major focus during the project; ensuring the user's data was secure was vital to the viability of this project. This was achieved through secure encryption and data verification with unique digital signatures. Using Next JS and Gun JS provided a positive developer experience, thanks to the monolithic architecture and developer-friendly features of both frameworks. To summarise, the integration of these two technologies resulted in a decentralized content management system that combines the flexibility of modern web frameworks with the resilience and transparency of decentralized storage.

Chapter 9. References

- A comprehensive survey on Blockchain-Based Decentralized Storage Networks. (2023). In <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10026822>
- Arcana Network. (2023, December 8). *What is Decentralized Cloud Storage and How does it Work?* [What is Decentralized Cloud Storage and How does it Work? | by Arcana Network | Medium](#)
- Blockchain-based decentralized storage networks: A survey. (2020). In <https://www.sciencedirect.com/science/article/abs/pii/S1084804520301302>
- Chamria, R. (2024, April 30). *Significant use cases of blockchain in decentralized storage. Blockchain Deployment and Management Platform | Zeeve.* <https://www.zeeve.io/blog/use-cases-of-blockchain-in-decentralized-storage/>
- From content management to enterprise content management. (n.d.). In <https://dl.gi.de/server/api/core/bitstreams/91e426ed-6bc8-4be5-9101-a980c08ab198/content>
- Gagan. (2023, February 21). *Why Decentralized CMS is the Future of Content Management for Web 3.0 and Beyond. Publive Blog.* <https://blog.thepublive.com/cms/why-decentralized-cms-is-the-future-of-content-management-for-web-3-and-beyond>
- GUN — the database for freedom fighters - Docs v2.0. (n.d.). <https://gun.eco/docs/Installation#server>
- Immutability and Decentralized Storage: An analysis of Emerging threats. (2019). In <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8941045>
- Irfan Khalid, M. (n.d.). *IEEE Xplore Full-Text PDF:* <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10026822>
- IPFS Documentation | IPFS Docs. (n.d.). <https://docs.ipfs.tech/>
- Jones, S. (2024, August 5). *6 Different types of content management systems. MadCap Software.* <https://www.madcapsoftware.com/blog/types-of-content-management-systems/>
- Moore, J. (2023, September 21). *blockchain storage. Search Storage.* <https://www.techtarget.com/searchstorage/definition/blockchain-storage>
- Next.js by Vercel - the React framework. (n.d.). <https://nextjs.org/>
- Öfverstedt, L. (2018). *Why Go Headless – A comparative study between traditional CMS and the emerging headless trend. In* <https://www.diva-portal.org/smash/get/diva2:1206058/FULLTEXT01.pdf>
- Osman, M. (2024, August 27). *What is a headless CMS? We'll explain. Blog LIVE.* <https://www.wix.com/studio/blog/headless-cms>

Pinto, R. (2020, February 21). *Costs of storing data on the blockchain*.

<https://www.1kosmos.com/blockchain/cost-of-storing-data-on-the-blockchain/>

Shadcn. (n.d.). *shadcn/ui*. *Shadcn/Ui*.

<https://ui.shadcn.com/>

Use MetaMask SDK with React UI | MetaMask developer documentation. (n.d.).

<https://docs.metamask.io/wallet/connect/metamask-sdk/javascript/react/react-ui/>

Who, what, and types of content management systems? (n.d.).

<https://www.oracle.com/ie/content-management/what-is-cms/>

Chapter 10. Appendix

Decentralized CMS Repository

<https://github.com/AdamGallagher27/Y4-Final-Project>

External Todo App

<https://github.com/AdamGallagher27/Y4-Todo>

External Hospital App

<https://github.com/AdamGallagher27/Y4-Hospital>

Figma Design file

<https://www.figma.com/design/paDUSko74HXXJbEI0C5Rpv/Final-Figma?t=M3OjBx83fOnJbL4k-1>