



# Secure Prescriptions System

Liam Ronan (Student) N00212101

WORD COUNT: 25875

SUBMISSION DATE: 29/04/2025

CHARACTER COUNT: 190698

FILE NAME: LIAM-RONAN-THESIS

# Secure Prescriptions System – A web application to allow doctors to send prescriptions securely to pharmacists

Author: Liam Ronan

Student Number: N00212101

Supervisor: John Montayne

Second Reader: Cyril Connolly

Frontend code: <https://github.com/LiamRonandev/MajorProjectFrontend>

Backend code: <https://github.com/LiamRonandev/MajorProjectBackend>

Date: 30/04/2025



Thesis submitted in partial fulfilment of the requirements for the BSc (Hons) in Creative Computing at the Institute of Art, Design and Technology (IADT)

## Declaration of Authorship

I hereby certify that the material which I now submit for assessment is entirely my own work and has not been taken from the work of others except to the extent of such work which has been cited and acknowledged within the text of my own work.

### **Declaration**

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

Signed: *Liam Ronan*

Date: 30/04/2025

## Abstract

This project explores the development of a secure, cloud-based prescription management system designed for doctors and pharmacists. It begins by looking into current solutions on the market, identifying key pain points for users, and then planning a system that could solve those issues in a practical and efficient way. The main focus throughout was ensuring robust security measures while enabling a good user experience. In parallel, the project explores industry best practices around cloud infrastructure, CI/CD pipelines and code quality. Some of the main features include role-based access control, multifactor authentication, data encryption, and real time notifications. The system allows doctors to manage digital prescriptions and patient information, while pharmacists can update statuses and leave notes. Its deployed using various AWS services such as EC2, S3, CloudFront, Route 53, and more. The end result is a modern, reliable, and scalable solution that makes prescription management more efficient, secure, and user friendly.

# Acknowledgements

I would like to thank my project supervisor, John Montayne, for his excellent guidance and encouragement during this project.

John's technical expertise, encouragement, and willingness to push me beyond my comfort zone greatly enhanced the system. His constructive criticism and strategic thinking kept me focused on realistic, user-centered solutions and inspired me to achieve my full potential throughout the project.

I also thank Dan, a practicing pharmacist, for testing the PharmaLink app and providing real-world feedback.

I'd like to thank my IADT lecturers over the past four years. Their teaching, guidance and support gave me the technical and professional skills to finish this project and prepare for a software development career.

Finally, I want to thank my family for their unwavering support, encouragement, and patience during my time in college.

## Table of Contents

Declaration of Authorship.....	2
Abstract.....	3
Acknowledgements .....	4
Table of Figures .....	9
1 Introduction .....	13
2 Research.....	14
2.1 Introduction.....	14
2.1.1 Security in Software Development.....	14
2.1.2 Importance of Security in Software Development .....	14
2.1.3 Challenges in Implementing Security in the SDLC.....	15
2.2 Security Challenges related to Personal Data .....	16
2.2.1 Unique Security Challenges in Handling Personal Data.....	16
2.2.2 GDPR and Compliance Considerations.....	17
2.3 Code Quality, Testing, and CI/CD for Secure Healthcare Systems .....	18
2.3.1 Code Quality and Security in Healthcare Systems .....	18
2.3.2 Testing for Security and Compliance .....	20
2.3.3 CI/CD Pipelines for Secure Development .....	21
2.4 Conclusion .....	23
3 Requirements Analysis .....	24
3.1 Introduction.....	24
3.2 Existing Applications .....	24
3.2.1 PioneerRX Pharmacy Software .....	24
3.2.2 WellSky Medication Management.....	26
3.4 User Personas .....	28
3.5 Use Case Diagrams.....	30
3.6 Requirements .....	31
3.6.1 User Requirements.....	31

3.6.2 Technical Requirements .....	32
3.6.3 Functional Requirements.....	33
3.6.4 Nonfunctional Requirements .....	33
3.7 Technical Feasibility Study .....	34
3.7.1 Technology Stack Selection .....	34
3.7.2 System Architecture .....	35
3.7.3 Technical Challenges and Mitigations.....	36
3.7.4 Cloud Infrastructure .....	37
3.7.5 Development Environment – Security, Code Quality, Testing.....	40
3.7.6 Continuous Integration & Continuous Deployment .....	42
3.8 Conclusion .....	45
4 Design .....	46
4.1 Introduction.....	46
4.2 System Architecture.....	46
4.3 Application Design .....	47
4.3.1 Technologies .....	47
4.3.2 Design Patterns .....	51
4.3.3 Database Design .....	53
4.3.4 Process Design.....	56
4.4 User Interface Design.....	58
4.4.1 Wireframes .....	58
4.4.2 Design System.....	59
4.5 Conclusion .....	60
5 Implementation.....	61
5.1 Introduction.....	61
5.2 Development Environment .....	61
5.3 Database.....	63
5.4 Cloud Infrastructure.....	65

5.4.1 Overview .....	65
5.4.2 Backend .....	65
5.4.3 Frontend .....	70
5.4.4 Summary .....	74
5.5 Continuous Integration & Continuous Deployment.....	75
5.5.1 Overview .....	75
5.5.2 Backend .....	75
5.5.3 Frontend .....	82
5.6 Development .....	89
5.6.1 Backend .....	89
5.6.2 Frontend .....	122
6 Testing .....	140
6.1 Introduction.....	140
6.2 Pharmacist Testing.....	140
6.2.1 Pharmacist Feedback .....	140
6.2.2 Analysis & Future Improvements .....	143
6.3 Front End Testing.....	144
6.3.1 End-to-End Testing .....	145
6.3.2 Vitest Unit testing .....	148
6.4 Backend Testing.....	151
6.4.1 Unit Testing with Jest.....	151
6.5 Conclusion .....	152
7 Project Management .....	153
7.1 Introduction.....	153
7.2 Project Phases .....	153
7.2.1 Proposal.....	153
7.2.2 Requirements.....	154
7.2.3 Design.....	154



7.2.4 Implementation .....	155
7.2.5 Testing .....	156
7.3 Project Management Tools .....	156
7.3.1 GitHub .....	156
7.3.2 Notion.....	158
7.4 Reflection .....	159
7.4.1 Personal Overview .....	159
7.4.2 Project Development .....	160
7.4.3 Project Oversight and Supervisor Communication .....	160
7.4.4 Technical Skills .....	161
7.4.5 Further Competencies and Professional Skills .....	162
7.5 Conclusion .....	162
8 Conclusion .....	163
References .....	165

# Table of Figures

Figure 1 PioneerRx Screen	25
Figure 2 PioneerRx Screen	26
Figure 3 WellSky Screen 1	27
Figure 4 WellSky Screen 2	27
Figure 5 User Persona 1	28
Figure 6 User Persona 2	29
Figure 7 Use Case Diagram 1	30
Figure 8 Use Case Diagram 2	31
Figure 9 MERN	35
Figure 10 Architecture	36
Figure 11 Basic Pipeline	43
Figure 12 System Architecture	46
Figure 13 Frontend Tech	47
Figure 14 MongoDB	50
Figure 15 REST API	53
Figure 16 ERD	55
Figure 17 Sequence Diagram 1	56
Figure 18 Sequence Diagram 2	57
Figure 19 Flow Chart 1	58
Figure 20 Flow chart 2	58
Figure 21 Authentication Designs	59
Figure 22 Dashboard designs	59
Figure 23 Design system	60
Figure 24 Insomnia	62
Figure 25 VS Code	63
Figure 26 MongoDB Atlas	64
Figure 27 Cloud Infrastructure	65
Figure 28 EC2 Security Groups	66
Figure 29 SSH PEM Key	66
Figure 30 NGINX Config	67
Figure 31 Application Load Balancer	68
Figure 32 API SSL Cert	69
Figure 33 Domain Names	69
Figure 34 S3 Buckets	70
Figure 35 S3 Latest Files	71
Figure 36 CloudFront Distribution	71

Figure 37 Cloud Invalidations	72
Figure 38 CloudFront Web Application Firewall	72
Figure 39 Lambda@edge Security Function	73
Figure 40 Security Pipeline	76
Figure 41 Snyk Dashboard	77
Figure 42 SonarCloud Dashboard	78
Figure 43 CI Pipeline	79
Figure 44 Deployment Pipeline	81
Figure 45 E2E Testing Pipeline	83
Figure 46 CI Pipeline	86
Figure 47 Deployment Pipeline	87
Figure 48 Backend Folder Structure	90
Figure 49 Prescription Model	92
Figure 50 Item Model	93
Figure 51 Database Connection Function	94
Figure 52 Application Starting Point	95
Figure 53 Server.js	96
Figure 54 env Example	96
Figure 55 User Model	97
Figure 56 Patient Model	98
Figure 57 Prescription Model	99
Figure 58 Medication Model	100
Figure 59 Item Model	101
Figure 60 Appointment Model	102
Figure 61 Create Prescription Function	103
Figure 62 Get Prescriptions	104
Figure 63 Pharmacist Update Function	105
Figure 64 Pharmacist Update Item Note	106
Figure 65 Prescription Route	107
Figure 66 Ensure Authenticated Middleware	108
Figure 67 Access Control Middleware	108
Figure 68 Verify Ownership Middleware	109
Figure 69 Validation Middleware	110
Figure 70 Error Middleware	110
Figure 71 Create JWT Function	111
Figure 72 Hashing Functions	111
Figure 73 Login Function	112
Figure 74 MFA Variables	113

Figure 75 Session Cookie	114
Figure 76 Verify TOTP Function	114
Figure 77 Patient Model 2	116
Figure 78 Encryption Key	116
Figure 79 Encrypt function	117
Figure 80 Decrypt Function	118
Figure 81 Socket.io Connection	119
Figure 82 Doctor Notification	119
Figure 83 Send Email Function	120
Figure 84 Medication Seeding	122
Figure 85 Frontend Project Structure	123
Figure 86 UI Components	125
Figure 87 Conditional Rendering	126
Figure 88 Inline Row Actions	127
Figure 89 Routes	127
Figure 90 __root Layout	128
Figure 91 Public Login Page	129
Figure 92 Dashboard Delete Mutation	130
Figure 93 React Hook Form Usage	130
Figure 94 Zod Validation Schema	131
Figure 95 Async Medication Select	132
Figure 96 usePrescriptions Hook	133
Figure 97 useMutation	134
Figure 98 OTP API Function	134
Figure 99 useQuery Auth	135
Figure 100 AuthContext	135
Figure 101 useAuth Hook	136
Figure 102 useAuth Usage	136
Figure 103 Quick Prescription Button	136
Figure 104 Socket Initialisation	137
Figure 105 Socket User Roles	138
Figure 106 Socket State	138
Figure 107 New Prescription Notification Function	138
Figure 108 useSocket Hook	138
Figure 109 Displaying Notification	139
Figure 110 Pharmacist Feedback	142
Figure 111 Pharmacist Issues Solved	144
Figure 112 E2E Tests Passing	145

<i>Figure 113 Firefox &amp; Chrome Tests</i>	146
<i>Figure 114 Navigation Test</i>	147
<i>Figure 115 Login Testing Setup</i>	148
<i>Figure 116 Prescription Unit Test</i>	149
<i>Figure 117 React Hook Unit Test</i>	150
<i>Figure 118 Unit Tests Passing</i>	150
<i>Figure 119 Jest Unit Tests Passing</i>	151
<i>Figure 120 JWT &amp; Hashing Tests</i>	152
<i>Figure 121 GitHub Repository</i>	157
<i>Figure 122 GitHub Actions Workflows</i>	158
<i>Figure 123 Notion Tasks</i>	159

# 1 Introduction

The project aims to develop a secure, cloud-based web application for doctors and pharmacists to track prescriptions. The project attempts to be user-friendly, emphasise secure data, and real time notifications for doctors and pharmacists. The doctors may create prescriptions whilst the pharmacist can update the status and notes of a prescription they were assigned to. The app will require the user to setup multi factor authentication and certain resources are only visible to specific roles. The system was developed using the MERN stack and AWS infrastructure.

## 2 Research

### Investigating Security and the Software Development lifecycle of healthcare systems.

#### 2.1 Introduction

Adding security to the Software Development Lifecycle (SDLC) is necessary to make systems that are strong and resilient. It is important to do this in areas like healthcare that deal with private information and important tasks. In this study, problems that can happen and ways to avoid them are looked at along with the reason why adding security to software development is important. This part talks about how to make systems that are safe and useful by looking at best practices for security, compliance, and new tools.

##### 2.1.1 Security in Software Development

##### 2.1.2 Importance of Security in Software Development

Security plays a huge role in software development it's what protects systems from potential threats, data breaches, and the kind of vulnerabilities that can lead to serious issues down the line. If security is ignored during the Software Development Lifecycle (SDLC), it can result in costly fixes, data loss, or even major system compromises. Assal and Chiasson (2018) highlight how this is often the result of rushed deadlines or gaps in developer knowledge, and they emphasise the importance of thinking about security from the very beginning of a project.

As systems become more complex, identifying risks early becomes even more important. Khan et al. (2022) explain that dealing with security concerns early in the SDLC not only reduces the number of vulnerabilities but also helps avoid the higher cost of fixing issues later. By implementing security into the process from the start, teams can avoid problems before they become serious.

Developers need the right training and tools to properly handle secure coding practices. Khair (2018) argues that secure development should be part of every stage of the SDLC, helping teams understand common vulnerabilities and how to prevent them. This approach improves

overall code quality, keeps systems compliant with regulations, and builds more reliable software.

Altogether, the research points to a clear conclusion: security shouldn't be treated as an afterthought. By making it a core part of the development process from requirements all the way through to deployment teams can build software that's both safe and effective. This not only protects users but also supports the growing need for transparency and trust in digital systems.

### 2.1.3 Challenges in Implementing Security in the SDLC

Theurich et al. (2023) say that implementing measures like threat modeling can be hard because team members aren't always experienced, there are tight deadlines, and it can be hard to find the right mix between being flexible and following strict security procedures. Because agile works in small steps, it might be hard to keep up the level of security research needed to find and fix problems as they arise.

People who are unwilling to change make these problems worse. ValdésRodríguez et al. (2023) say that the biggest problems with adding security to agile routines are limited organizational and procedural freedom, separate processes, and a lack of teamwork. Teams sometimes see security measures as getting in the way of production, putting more value on usefulness and speed of delivery than on long-term security and stability.

To get around these problems, you need an individualized approach. Theurich et al. (2023) say that security measures should be built into agile processes as essential parts of the work, not as extra chores. To create a mindset of shared security duty, departments can work together and developers can get focused training to fill in any gaps in their knowledge. ValdésRodríguez et al. (2023) say that adopting this way of thinking is important for dealing with security problems in a way that doesn't compromise the flexibility and efficiency that agile methods need.

To get around these problems, organizations need to come up with unique solutions that combine rapid ideas with security measures. For example, lightweight threat modeling methods like STRIDE or PASTA can be used during sprint planning meetings to make sure that security risks are identified and reduced without getting in the way of the development process. Theurich



et al. (2023) say that security jobs should be added to agile practices like standups and retrospectives so that security issues are always visible.

We also need better training and tools. You can help developers find and fix security holes by giving them tools like OWASP's Secure Coding Practices Guide or by making security learning sites like Secure Code Warrior a normal part of their work.

Companies can make software that is both flexible and reliable by making their security goals match the concepts of rapid development and continuous security integration.

## 2.2 Security Challenges related to Personal Data

### 2.2.1 Unique Security Challenges in Handling Personal Data

Specifically in the context of the big data lifecycle, managing personal data poses serious security issues. Data collection, storage, processing, and sharing are some of the stages that make up this lifecycle, and each one presents unique vulnerabilities. Inadequate protections or poorly secured systems can lead to breaches and illegal access at any point, presenting significant risks to both individuals and companies (Koo et al., 2020). These difficulties are only made worse by the growth dependence on data driven decision making, since vast amounts of personal data are now easy targets for criminals.

It is especially difficult to protect privacy while preserving data usability. Businesses need to find a balance between protecting private data and facilitating insights based on large databases. According to Kantarcioglu and Ferrari (2019), achieving this happy medium often means sacrificing security for scalability. For example, scalable Encryption techniques are necessary to safely protect data in large systems, but They may also slow down the efficiency and speed of data analytics. Strong access control measures must also be put in place to stop illegal use, but badly designed systems may unintentionally restrict the use of data that is approved or make processes more difficult.

Some of these issues may be resolved by emerging technology like privacy preserving

data analytics. Organisations can examine data without putting individual records at risk thanks to strategies like homomorphic encryption and differential privacy. According to Kantarcioglu and Ferrari (2019), incorporating these strategies into the data lifecycle is crucial for protecting privacy and guaranteeing adherence to changing legal requirements. However, large investments in infrastructure, knowledge, and development is necessary for the largescale adoption of such technologies.

Organisations must deal with more general systemic problems in addition to technological ones, like developing a security aware culture and coordinating procedures with legal requirements. For example, regular policy updates and security audits may help minimise the risks brought on by quickly evolving threats and technologies. A single approach to protecting personal data requires cooperation from All parties involved, including developers, data scientists, and legal teams.

Protecting personal data requires more than reactive measures as its use continues to grow. Organisations may manage the challenges of protecting personal data and create systems that inspire confidence by implementing innovative technologies, encouraging teamwork, and strictly adhering to legal requirements.

## 2.2.2 GDPR and Compliance Considerations

Software development processes face a difficult task when trying to comply with the General Data Protection Regulation (GDPR), especially when managing personal data. Principles like data minimisation, privacy by design, and the defence of individual rights are highlighted by the GDPR. It takes a good understanding of both technical implementation and legal requirements to incorporate these ideas into software development. NegriRibalta et al. (2024) emphasise the importance of requirements engineering in this procedure, contending that early in the SDLC, specific legal requirements must be translated into workable development tasks. However, knowledge gaps and the challenge of converting recommendations into practical engineering practices often make this difficult.

Putting in place efficient consent procedures is one common issue. According to Franke et al. (2024), several open source projects have inconsistent or inadequate

permission procedures that do not adhere to GDPR regulations. For example, non compliance may occur from unclear and difficult to use interfaces for gaining user consent, which could expose companies to penalties and harm the company. These problems are made worse by a lack of resources and experience, which makes it challenging for smaller teams to guarantee compliance in every area of their program.

Organisations must take a deliberate and methodical approach to GDPR compliance in order to overcome these challenges. NegriRibalta et al. (2024) advise that the SDLC incorporate GDPR principles immediately, starting with requirements gathering that is privacy focused. To make sure developers are aware of their compliance responsibilities, this involves working with legal and regulatory professionals. Development teams can also be less burdened by tools and frameworks that automate GDPR compliance tests, like confirming consent channels or evaluating data reduction techniques.

According to Franke et al. (2024), incorporating these procedures into team processes can lower the possibility of expensive oversights while also greatly improving compliance results.

## 2.3 Code Quality, Testing, and CI/CD for Secure Healthcare Systems

### 2.3.1 Code Quality and Security in Healthcare Systems

The safety and reliability of healthcare systems, which handle sensitive patient data and vital functions, depend on maintaining excellent code quality. In addition to making vulnerabilities more likely, poorly written or unmaintainable code makes debugging and Updating is more difficult and may compromise system functionality. Effective techniques like automated testing, secure coding standards, and static code analysis can greatly improve code quality and security, according to Java Tech Blog (2024).

Early in the development process, static code analysis tools such as SonarQube, Checkmarx and Fortify are vital for identifying coding discrepancies and security flaws. These tools check the source code for problems such as poor cryptographic

implementations, buffer overflows, and hardcoded credentials. By integrating these technologies throughout the development process, developers can reduce technical debt by identifying and correcting issues before they become more serious. Such proactive detection is essential for healthcare institutions, because hacks may reveal private medical information.

Writing robust, secure code is based on secure coding recommendations, like those offered by OWASP or CERT. Developers can steer clear of common dangers like SQL injection, inadequate input validation, and unsafe error handling by following these standards. According to Java Tech Blog (2024), implementing these standards into regular development procedures promotes a security culture and guarantees that teams give equal weight to functionality and resilience.

Other ways to improve code security and quality include pair programming and peer code reviews. By allowing several developers to review the same code, these procedures increase the possibility of finding mistakes or vulnerabilities that automated tools would miss. Peer evaluations are essential for upholding coding standards and encouraging team members to share information, according to Aptori.dev (2024).

In healthcare systems, technical debt results from hurried or inadequate coding. Code quality management is another area that requires careful management. Unmanageable, bloated codebases that are more vulnerable to security flaws can be the outcome of unpaid technical debt. Teams may measure and address technical debt with the aid of tools like SonarQube or Code Climate, which offer useful insights into areas that need refactoring. Reducing technical debt is a long term investment in system security and maintainability, according to Aptori.dev (2024).

Code quality is maintained in large part by automated testing, especially in the form of security, integration, and unit tests. For instance, unit tests that confirm that each system component works as intended can be made using tools like JUnit or TestNG. While security focused testing tools like OWASP SAP may imitate assaults on the application, integration testing tools like Selenium or Cypress can evaluate how components interact. According to Java Tech Blog (2024), adding these tests to the CI/CD pipeline guarantees that healthcare systems are resilient to security and functional failures.

In conclusion, healthcare systems need a comprehensive strategy for secure development that blends superior technical skills with a consistent dedication to security. Code quality must be given top priority by organisations through thorough testing, teamwork techniques like code reviews, and ongoing developer education. By implementing these precautions, healthcare software can satisfy the needs of both security and functionality, safeguarding patient information and preserving confidence in the systems that support vital healthcare services.

### 2.3.2 Testing for Security and Compliance

Software development must include security and compliance testing, especially in industries like healthcare where strict regulations and sensitive data are combined. Comprehensive testing guarantees that software systems continue to be safe from attacks and comply with applicable legal requirements. To proactively detect vulnerabilities, Potter and McGraw (2004) suggest that security testing should be beyond traditional functional testing. To identify vulnerabilities that attackers could exploit and enable teams to fix them prior to deployment, techniques like fault injection and penetration testing are essential.

Identifying vulnerabilities is only one aspect of security testing; another is foreseeing possible attack routes that may evolve over time. Understanding the attackers point of view is essential to doing effective security testing, say Potter and McGraw (2004). They highlight how crucial it is to replicate actual attack instances to find system vulnerabilities that can go undetected during regular functional testing. Techniques like fault injections, which intentionally introduce problems into the system, can expose hidden vulnerabilities and show how software responds under pressure.

Additionally, Potter and McGraw stress how important it is to use security testing to guide and improve the software architecture. For example, businesses might Proactively modify design concepts and coding techniques to prevent similar concerns in future iterations by recognising recurrent patterns of vulnerabilities. This architectural feedback loop improves the software's longterm resilience against changing threats in addition to its immediate security posture.

Selecting the highest risk areas for testing as a top priority is another important insight. Potter and McGraw support focused security testing that concentrates on components with the greatest possible impact in the case of a breach because not all system components need the

same level of attention. This could require making the testing of modules managing patient data encryption, authentication systems, and external API integrations, the highest priority for healthcare systems.

Finally, they note that security testing needs to cover the operational environment in which the product will operate in addition to code level testing. For instance, even while a system passes all internal security checks, it may not be able to fend against assaults that take advantage of third party dependencies or network configurations. A more comprehensive approach is ensured by integrating environmental aspects into security testing, which aligns the software's security posture with actual circumstances.

On the other hand, compliance testing makes sure that software systems adhere to legal regulations, as those set down by the General Data Protection Regulation (GDPR) or the Health Insurance Portability and Accountability Act (HIPAA). The significance of continuous compliance testing, which incorporates automated compliance checks into the development process, is pointed out by Moscher (2017). This ensures that, despite updates and alterations, systems maintain compliance over the course of their lifetime. Without depending entirely on manual audits, teams can find gaps in adherence to regulatory standards by automating compliance validation.

In both security and compliance testing, automation is critical. Automated testing tools can assess vulnerabilities, replicate real world attack scenarios, and confirm compliance with legal requirements. For instance, compliance tools can make sure that systems comply with privacy and data protection regulations, while penetration testing tools may imitate malicious assaults to find any vulnerabilities. Automation improves the dependability of the results by speeding up the testing process and lowering human error.

### 2.3.3 CI/CD Pipelines for Secure Development

For vulnerabilities to be found and fixed early in the software development lifecycle, security must be included in CI/CD pipelines. This requires a blend of complex setups, specialist tools, and careful pipeline architecture. Sydneyacademics.com (2024) underlines the importance it is to incorporate security technologies straight into CI/CD processes to enforce strict security guidelines without slowing down development.

Automated security testing tools are an essential part of secure CI/CD pipelines. During the building phase, code can be examined by Static Application Security Testing (SAST) tools to find vulnerabilities like SQL injection, cross site scripting (XSS), or unsafe function calls. Similarly, to find runtime vulnerabilities that static analysis could overlook, Dynamic Application Security Testing (DAST) technologies simulate assaults on the active application. For complete coverage, CI/CD pipelines usually incorporate tools like SonarQube, OWASP SAP, and Snyk. Mangla (2024) suggests the importance it is to automate these tools to impose uniform security standards and guarantee that each build is put through the same rigorous inspection.

Ansible and Terraform are two examples of configuration management tools that are essential to securing the deployment process. By defining infrastructure as code, these technologies enable teams to create secure and consistent setups across environments. Developers may avoid common configuration errors, including open ports or unsafe default settings, that could leave systems vulnerable to assaults by incorporating these tools into CI/CD pipelines.

Software composition analysis (SCA) tools such as Dependabot and WhiteSource come in handy for finding vulnerabilities in third party libraries and dependencies, in addition to vulnerability and configuration scanning. When opensource elements are used, these tools make sure that vulnerable or out of date packages are identified and fixed before deployment. This layer of automated dependency management lowers the risk of supply chain attacks, which are becoming more common, according to Sydneyacademics.com (2024).

Another crucial component of secure CI/CD pipelines is ongoing monitoring. To monitor pipeline activity and spot suspicious patterns, like unauthorised code or configuration changes, tools like Prometheus, Datadog, and Splunk can be integrated. Realtime warnings from these systems allow for quick reactions to threats.

Mangla (2024) recommends using cloud native security solutions to guarantee that Protection is effective in a variety of environments. Built in security services like Google Cloud Builds automated vulnerability scanning, Azure DevOps advanced compliance checks, and AWS Code Pipeline integrated security testing are all provided by cloud providers including AWS, Azure, and Google Cloud. Organisations can customise their

pipelines to meet the unique needs of their infrastructure by utilising these platform native solutions.

In conclusion, creating secure CI/CD pipelines requires the purposeful integration of modern tools, proper configurations, and strong monitoring systems in addition to a general dedication to security. Organisations may make sure that their pipelines not only produce secure software but also withstand changing threats by utilising automated testing, infrastructure as code, and dependency analysis.

## 2.4 Conclusion

This research emphasises the importance of security in the Software Development Lifecycle (SDLC) to create reliable systems, minimise risks, and protect sensitive data. Techniques include integrating automated tools, incorporating security into agile practices, and cultivating a collaborative culture. Prioritising safe coding, proactive threat modelling, and ongoing compliance testing ensures the security of sensitive data and critical infrastructure, promoting dependability and confidence in the digital environment.



## 3 Requirements Analysis

### 3.1 Introduction

The requirements analysis phase is an essential phase in ensuring that the system being created satisfies the expectations of its users. This phase involves a thorough examination of user expectations, system operation, and technical viability to provide a clear development path. By considering the needs of doctors and pharmacists, the system can be designed to provide a user friendly and efficient experience.

### 3.2 Existing Applications

Below are two examples of prescription management applications outlining a list of their features along with benefits and drawbacks.

#### 3.2.1 PioneerRX Pharmacy Software

A complete pharmacy management system called PioneerRx was created to improve several pharmacy activities.

**Key features:**

- **Medication therapy management (MTM):** Medication synchronisation and adherence monitoring.
- **Patient Risk Scores:** Evaluate and track health risks for patients to deliver individualised treatment.
- **Medication Synchronisation (Med Sync):** Aligns patient prescriptions to increase adherence and expedite refills.

**Benefits:**

- **Rich Feature Set:** Provides a variety of tools for efficient pharmacy operations management.
- **Customisable Interface:** Customisable interface enables pharmacies to adapt it to their unique workflows.
- **Emphasis on Patient Care:** Highlights resources that improve medication compliance and patient outcomes.

## Drawbacks:

- **Learning Curve:** Staff members may need some time and training to properly use the numerous capabilities.
- **Cost considerations:** Smaller pharmacies may find comprehensive systems prohibitively expensive.

**Enter Sales Items** Change Due Last Sale: \$0.00

Item:  Search for an item

Description:  Quantity:  Price:  Tax Rate:

Dept	Description	Qty	Price	Est price	Tax	Taxable
Rx	135562-00 - Pierre Cuisin	1.00	\$0.00	\$0.00	0.00%	N
Food & Bev	Candy Bar - Kit Kat	1.00	\$0.80	\$0.80	0.00%	N

**Summary:**

Count	2 (1 Rx)	Subtotal	\$0.00	Tax	\$0.00	Flax	\$0.00	Balance	\$0.00
-------	----------	----------	--------	-----	--------	------	--------	---------	--------

Figure 1 PioneerRx Screen

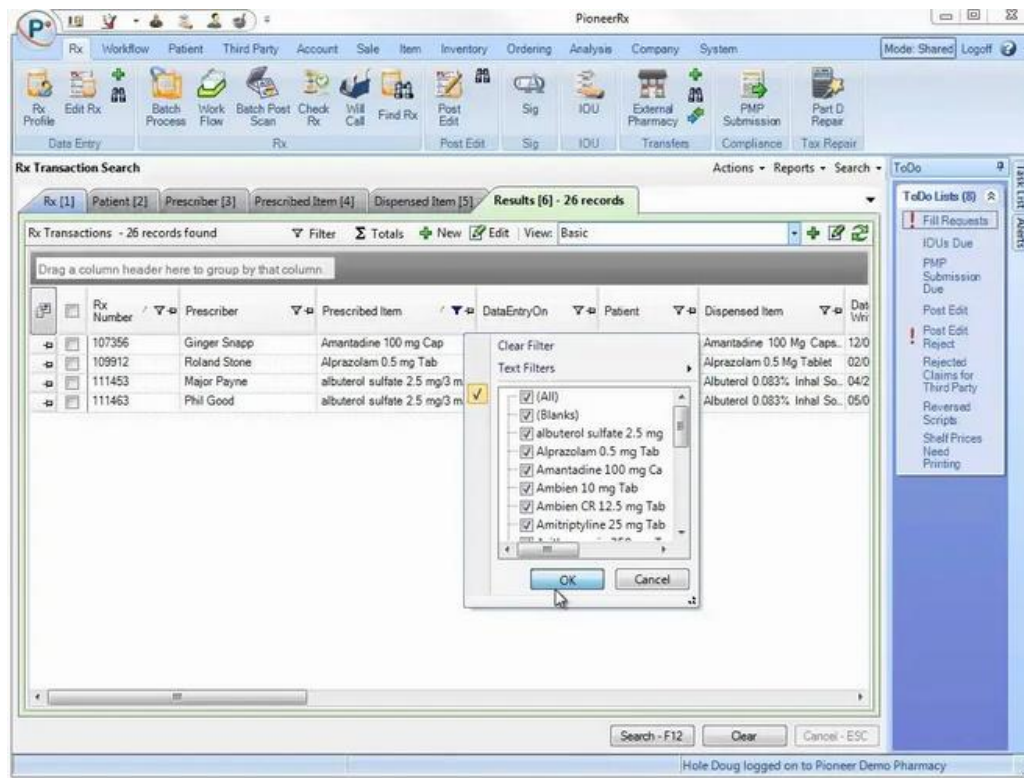


Figure 2 PioneerRx Screen

### 3.2.2 WellSky Medication Management

WellSky provides a drug management solution that automates and streamlines clinical operations.

#### Key Features:

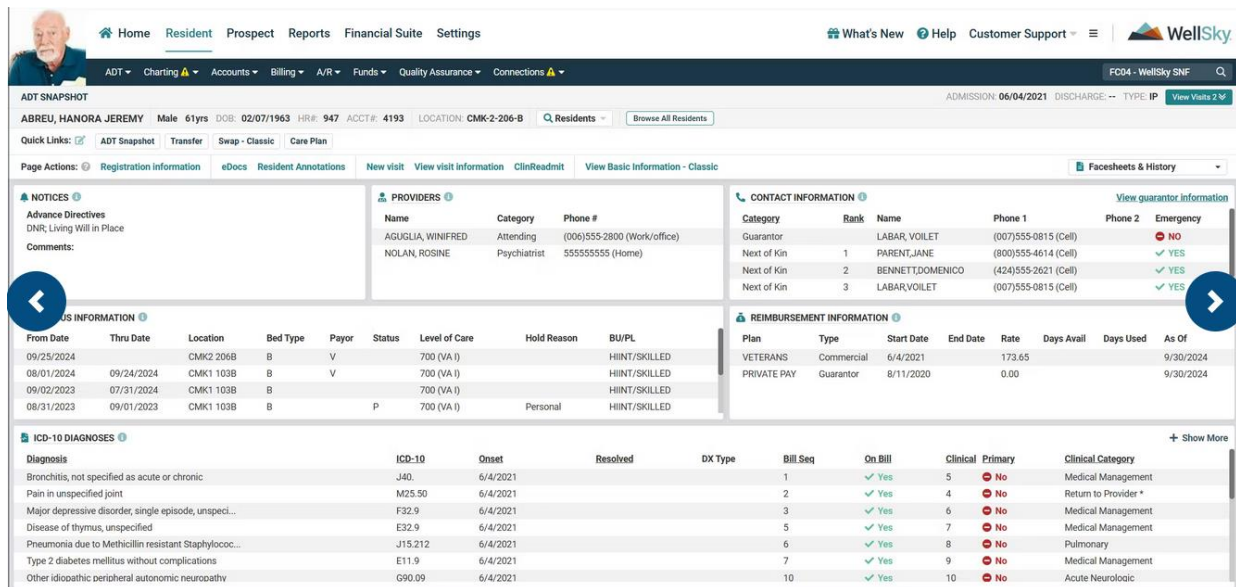
- **Clinical Workflow Automation:** Reduces manual labour and increases efficiency by streamlining procedures.
- **Error Reduction Tools:** Reduces drug distribution errors by implementing checks and balances.
- **Improvements to Patient Safety:** Offers resources to guarantee that patients receive their medications precisely.

#### Benefits:

- **Emphasis on Safety:** Puts patient safety first by minimising mistakes and managing medications precisely.
- **Efficiency Gains:** Workflow automation can result in more productivity and time savings.
- **Innovative Tools:** Uses modern features.

## Drawbacks:

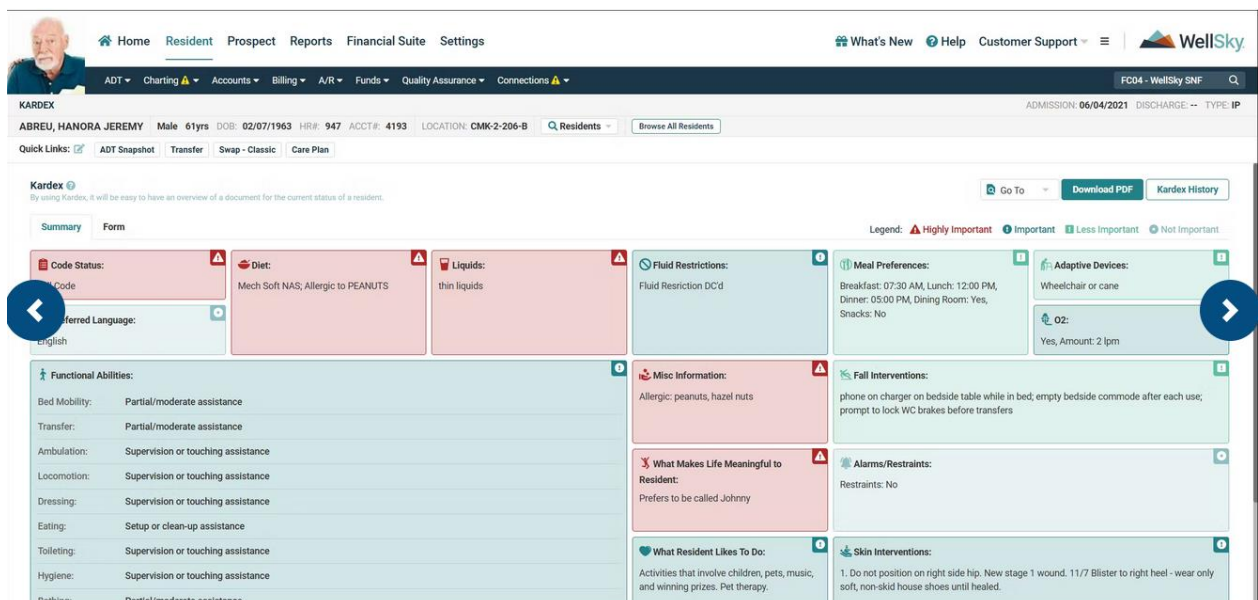
- **Integration Challenges:** During implementation, compatibility checks with current systems could be necessary.
- **Requirements for Training:** Employees may require training to become accustomed to new automated procedures and equipment.



WellSky interface showing resident information for ABREU, HANORA JEREMY. The screen displays various tabs including ADT Snapshot, Transfer, Swap - Classic, and Care Plan. Key sections include:

- NOTICES:** Advance Directives (DNR, Living Will in Place).
- PROVIDERS:** List of providers including AGUGLIA, WINIFRED and NOLAN, ROSINE.
- CONTACT INFORMATION:** Guarantor (LABAR, VOILET) and Next of Kin (PARENT, JANE; BENNETT, DOMENICO; LABAR, VOILET).
- REIMBURSEMENT INFORMATION:** Plan (VETERANS), Type (Commercial), Start Date (6/4/2021), End Date (8/11/2020), Rate (173.65), Days Avail (0.00), Days Used (9/30/2024).
- ICD-10 DIAGNOSES:** Table listing diagnoses such as Bronchitis, Pain in unspecified joint, Major depressive disorder, and others with their respective ICD-10 codes and onset dates.

Figure 3 WellSky Screen 1



WellSky interface showing the Kardex summary for ABREU, HANORA JEREMY. The screen displays various tabs including Summary and Form. Key sections include:

- Code Status:** Code (Circled in blue).
- Diet:** Mech Soft NAS; Allergic to PEANUTS.
- Liquids:** thin liquids.
- Fluid Restrictions:** Fluid Restriction DC'd.
- Meal Preferences:** Breakfast: 07:30 AM, Lunch: 12:00 PM, Dinner: 05:00 PM, Dining Room: Yes, Snacks: No.
- Adaptive Devices:** Wheelchair or cane.
- Functional Abilities:** Table listing abilities such as Bed Mobility, Transfer, Ambulation, Locomotion, Dressing, Eating, Toileting, Hygiene, and Bathing.
- Misc Information:** Allergic: peanuts, hazel nuts.
- Fall Interventions:** phone on charger on bedside table while in bed; empty bedside commode after each use; prompt to lock WC brakes before transfers.
- What Makes Life Meaningful to Resident:** Prefers to be called Johnny.
- Alarms/Restraints:** Restraints: No.
- What Resident Likes To Do:** Activities that involve children, pets, music, and winning prizes. Pet therapy.
- Skin Interventions:** 1. Do not position on right side hip. New stage 1 wound. 11/7 Blister to right heel - wear only soft, non-skid house shoes until healed.

Figure 4 WellSky Screen 2

## 3.4 User Personas

User Personas are a fictional, yet realistic depiction of a character that represent the traits of the target audience. Developers can have a grasp of the requirements, habits, and preferences of the audience by creating user personas. (NN Group, 2015)

Personas also serve as a basis for making well informed judgments at every stage of the development process, from UI/UX design choices to feature priority. Teams can effectively reduce any risks and difficulties during the development process by using personas. (M. Chu, 2023).

Below in figures 5 and 6, developed user personas for doctor and pharmacist users by outlining their work preferences, needs & goals, and pain points.

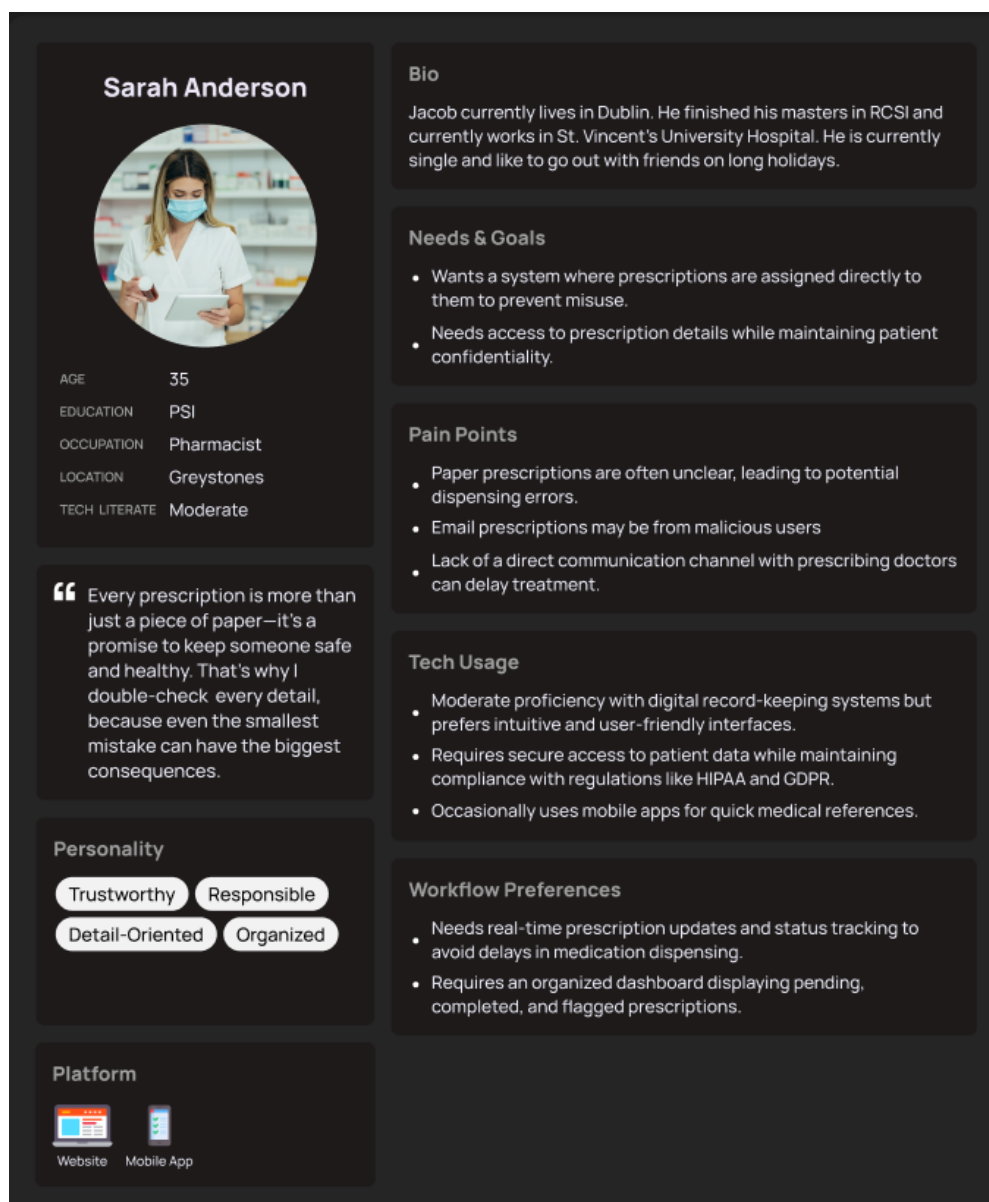


Figure 5 User Persona 1

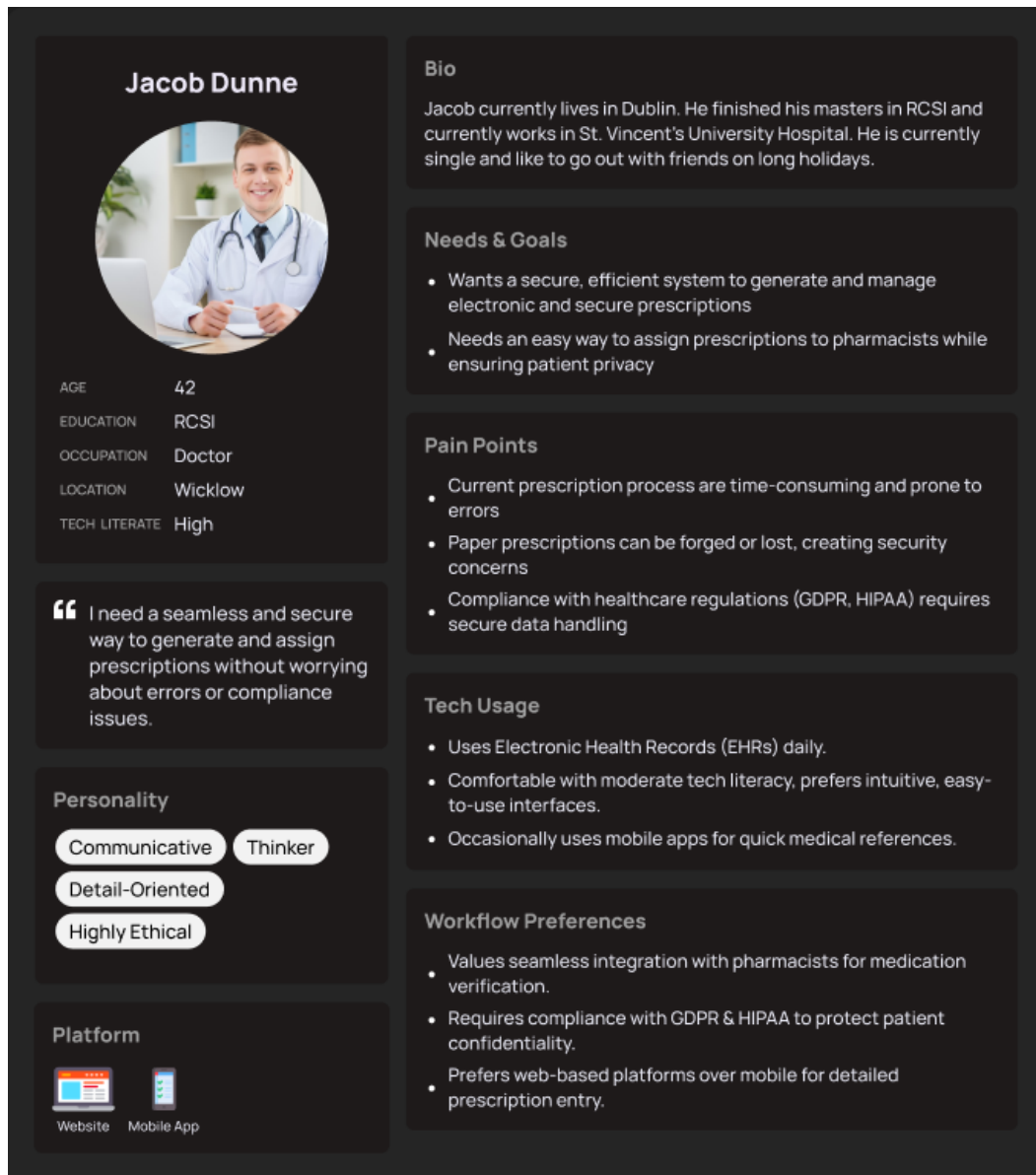


Figure 6 User Persona 2

### 3.5 Use Case Diagrams

Use case diagrams were created to visualise a high-level representation of the system to illustrate the interactions between the users (actors) and a system. They capture the functional requirements of the system, showcasing how the different users engage with various use cases, or specific functionalities within the system. (GeeksForGeeks, 2025)

The first diagram illustrates how a user will create an account or login to the system, enabling multi-factor authentication with an authenticator app of their choice

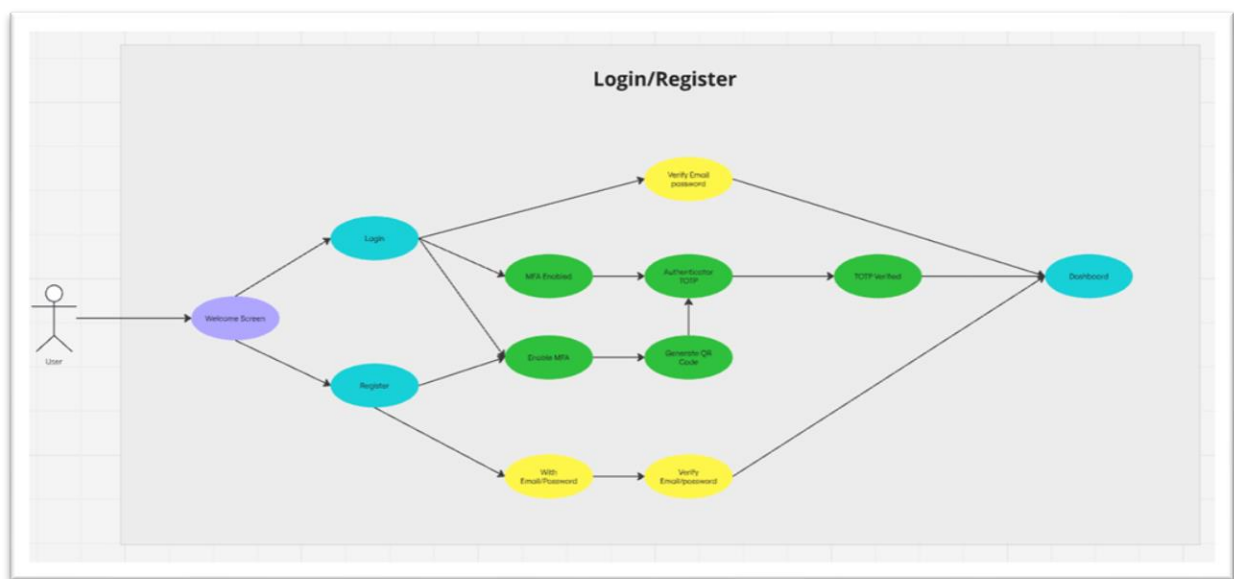


Figure 7 Use Case Diagram 1



The diagram in figure 8 below outlines how a doctor or a pharmacist may navigate through the application. The ovals in blue contain core functionality and features whereas the ovals in white are extended checks.

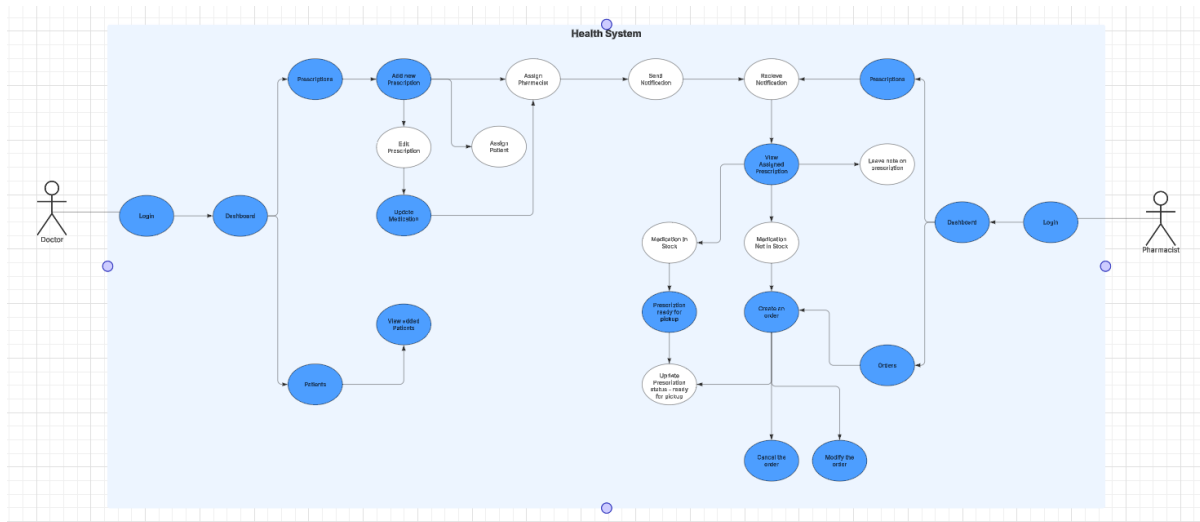


Figure 8 Use Case Diagram 2

## 3.6 Requirements

For it to make sure that the finished product satisfies business objectives, and technical feasibility, requirements gathering is a key process in the software development lifecycle (SDLC). Functional and nonfunctional needs must be gathered, examined, verified, and documented. Project failures, rework, cost overruns, and misplaced expectations can result from poor requirements engineering. Clear requirements allow development teams to produce solutions that satisfy customer expectations, enhance user experience, and guarantee regulatory compliance in industries including cybersecurity, finance, and healthcare. After the completion of the user personas and use case diagrams, created a list of the various types of requirements below. Below will go into more detail for each of the categories.

### 3.6.1 User Requirements

The User requirements ensure that the software system meets its intended users' specific needs, expectations, and goals. Understanding and effectively managing these user requirements is essential.

- Users should be able to register with email and password and select a role such as doctor or pharmacist.



- MFA Will be enabled by default and will generate a QR code that they can scan with their authenticator app such as Microsoft Authenticator to create a TOTP.
- Upon logging in as a Doctor, they can add new patients to the system, create new prescriptions and assign a pharmacist and patient to the prescription.
- A doctor may be allowed to update a patient with new emergency contact details, address, DOB etc.
- A doctor may remove a patient from the system which will also remove all their prescriptions.
- Upon logging in as a pharmacist, they may view all the prescriptions that they were assigned to from a doctor.
- A pharmacist should be notified when a new prescription is assigned to them – notification alert.
- A doctor should be notified when there is status change on a prescription.
- A pharmacist may update the status of a prescription such as pending, processed, completed, cancelled etc and they may leave a note on the prescription such as “Ordering medication”.
- A pharmacist can view patients through the prescription they were assigned to from the doctor.
- The server generates a JSON web token and stores it in secure, HTTPOnly cookie.
- Doctors should be able to select medication for prescriptions from a real list of verified and marketed medications in Ireland
- Logout of their account.

### 3.6.2 Technical Requirements

Below will be a blueprint that outlines the functionalities, features, and technical aspects of this software system. These requirements will outline how the technical aspects will function and interact with one another.

- Will it be technically feasible to develop the full stack application.
- The system should be highly secure as it engages with sensitive patient and prescription data.
- Upon registering, the administrator should receive an email with the users credentials.
- The system should be developed with software development best practices in mind such as CI/CD, security, testing.

- The system should be fast and a smooth user experience.
- The system should be deployed safely and securely using AWS Cloud infrastructure.
- The system should encrypt passwords and GP/pharmacist license numbers.
- The system will follow Authentication best practices such as JWTs, encryption, MFA, HTTP only cookies.
- The system should be end to end tested and have sufficient component unit tests on the frontend and sufficient unit tests on the backend
- The system should be constantly scanned for security and code quality issues using Snyk and SonarQube.
- The system should use real medication data.

### 3.6.3 Functional Requirements

Functional requirements are services or components the system must deliver. The functional components are the features and functions that the developers must implement to enable the users to accomplish their tasks (Altexsoft, 2023). Below is a list of features to develop with the first feature the most important and critical to implement

1. The users should be able to successfully log in and create accounts securely
2. Allow doctors to create a prescription and assign a pharmacist and a patient
3. Allow doctors to modify, or edit a prescription for the pharmacist
4. Encrypt the sensitive patient and prescription data
5. Allow doctors to add patients to the system and view all their medical history, prescriptions, personal details
6. Allow pharmacist to update a prescription with a new status

### 3.6.4 Nonfunctional Requirements

A set of specifications that describe the systems operation capabilities and constraints. These requirements are to outline how well the system operates, including speed, security, reliability, and data integrity. If these specs were not met, it could result in the system not performing as well as it should.

- The frontend application should be a smooth and accessible user interface.
- Should have fast load times navigating through pages and should not take too long to modify or create a resource.
- Patient and prescription data should be securely encrypted.
- Should be able to scale with large volumes of traffic.

- Should be available to users of various regions.
- The design should follow a design system with consistency.

## 3.7 Technical Feasibility Study

There are various technologies and languages that could be used to develop this system; however, the secure prescriptions system is being developed using the MERN (MongoDB, Express, React, Node) stack. Below will delve into more detail around the specifics of the technologies used

### 3.7.1 Technology Stack Selection

- **Frontend:** React with TypeScript for a type safe and scalable user interface
- **Backend:** NodeJS with express.js for developing the REST API
- **Database:** MongoDB atlas for flexible and scalable NoSQL data storage
- **Authentication & Security:** JWT for auth, multifactor authentication using time based onetime passwords for enhanced security, Node native crypto module for encrypting and decrypting sensitive patient and prescription data
- **Testing:** Vitest Unit tests and Playwright end to end test for the frontend and Jest unit tests for the backend
- **Realtime notifications:** Socket.io for pharmacist notifications
- **Continuous Integration:** GitHub Actions for running tests, linting, security and code quality scanning, formatting
- **Continuous Deployment:** AWS S3 Bucket and CloudFront for the frontend deployments and an EC2 Instance with a Nginx reverse proxy server for the API backend



Figure 9 MERN

### 3.7.2 System Architecture

System architecture explains the systems core ideas and characteristics regarding its relationships, environment, and other design principles. The architecture includes the organisational structure, behavioural components, and the composition of those components into more complex subsystems. (Gupta, R, 2024).

#### 3.7.2.1 Three tier Architecture

The system being developed follows a three-tier architecture, a widely adopted design pattern that separates the application into three layers.

- Presentation layer (Frontend): React.js is used by the frontend to provide an intuitive user interface, and HTTPS is used to securely communicate data with the backend REST API. It manages authentication, prescription, patient, and responsive design across platforms.
- Business logic layer (Backend REST API): NodeJS Express is used by the backend REST API for data processing, authorisation, and authentication. The API is protected by HTTPS, validation, and auth middleware and will adhere to RESTful standards. It stores and retrieves data by interacting with a MongoDB database.
- Data Layer (Database & Storage): MongoDB Atlas is used by the data persistence layer to store encrypted user credentials, patient information, and medications in a cloud managed NoSQL database. To avoid unwanted access, the system has role based access control in and backup capabilities.

Basic diagram outlining a high-level overview of the system architecture as seen in figure 10:

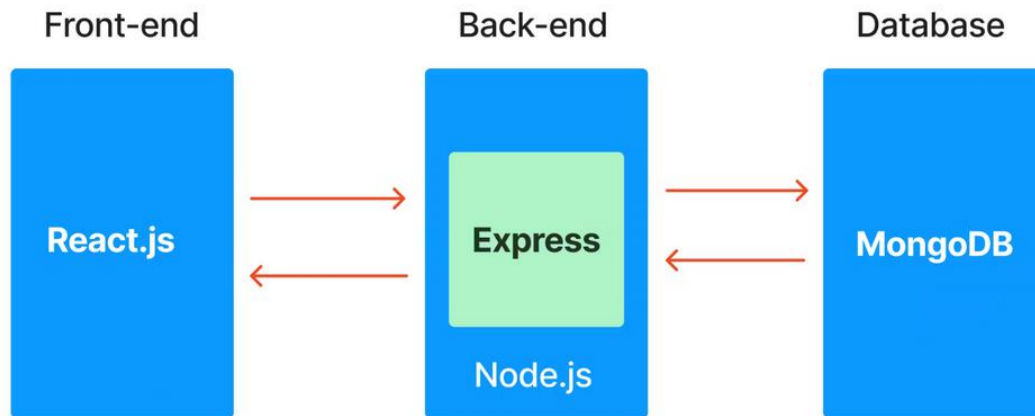


Figure 10 Architecture

### 3.7.2.2 Advantages of Three tier Architecture

There are various advantages to the three tier architecture:

- **Separation of concerns:** The system is simpler to expand and maintain since each layer concentrates on a different component.
- **Scalability:** Due to its decoupling, each layer may be grown separately to deal with growing traffic.
- **Security:** There is less chance of data exposure because sensitive processes like encryption and authentication are only managed on the backend.
- **Improved Maintainability:** Faster development cycles are made possible by the fact that code changes made to one layer won't immediately affect the other layers.

Using ALB for traffic control, CloudFront for content distribution, Route 53 for DNS resolution, and EC2 for backend hosting, the architecture is deployed on AWS. These cloud based solutions ensure the systems high availability, scalability, and security.

### 3.7.3 Technical Challenges and Mitigations

Below is list of the most important challenges for the application and their mitigation strategies

Challenges	Mitigation Strategy
Data Security & Encryption	Use AES256 encryption for sensitive patient & prescription data, HTTPS/TLS for secure communication
Scalability	AWS EC2 autoscaling, AWS CloudFront to serve and protect the static assets in the S3 bucket

Realtime notifications	Socket.io for realtime notifications
Role Based Access Control	Implement RBAC to restrict access to certain routes in the application for both doctor & pharmacist.
High Availability	An Application Load Balancer in front of the Nginx proxy server to ensure uptime
Frontend performance	Optimise React with lazy loading, memoisation, efficient state management

### 3.7.4 Cloud Infrastructure

Cloud infrastructure and deployment are fundamental for modern software systems since it offers applications scalability, flexibility, and security. Cloud based deployment makes sure fault tolerance and high availability while reducing operational expenses by removing the need for on premises equipment.

Using several AWS services to manage frontend, backend, networking, security, and data storage, the cloud infrastructure for this system is meant to be scalable, safe, and very available.

#### Amazon S3 – Static Asset storage for Frontend

Amazon S3, a robust and scalable object storage solution, will host the React frontend. S3 offers a serverless and affordable solution for hosting static files, including photos, HTML, CSS, and JavaScript.

##### Requirements:

- AWS restrictions are used to limit public access to CloudFront exclusively.
- Lifecycle rules and versioning for effective storage management.

##### Benefits:

- Benefits include durability and high availability (99.9% uptime).
- Delivery of content with low latency when combined with CloudFront CDN.
- Scalable storage that doesn't require human involvement.

#### Amazon CloudFront – Content Distribution

Amazon CloudFront is used as a Content Delivery Network (CDN) to distribute frontend content globally, ensuring low latency access for users. It caches static assets from S3 across edge locations, improving performance and reducing load times.

**Requirements:**

- CloudFront distribution set up with S3 as the origin.
- TLS/SSL configuration using AWS Certificate Manager (ACM) for secure HTTPS access.
- Lambda@Edge function to handle security headers.

**Benefits:**

- Faster content delivery via edge caching.
- DDoS protection via AWS Shield integration.
- Improved security with signed URLs and HTTPS enforcement.

## Amazon Route 53 – Domain name management

Amazon Route 53 is a scalable and highly available DNS (Domain Name System) service that will manage the custom domain (healthservice.click) for the system.

**Requirements:**

- Custom domain name configuration for the front end and API.
- DNS records route traffic to CloudFront (frontend) and ALB (backend API).
- SSL/TLS security integrated via ACM.

**Benefits:**

- Low latency DNS resolution with high availability.
- Automatic failover in case of service outages.
- Easy integration with AWS services for seamless routing.

## AWS Web Application Firewall (WAF) – Security Layer

AWS WAF will be used to protect the frontend and backend from common web attacks, such as SQL injection, XSS, and bot attacks.

**Requirements:**

- WAF rule set to filter malicious traffic.
- Integration with CloudFront for frontend protection.
- IP blocking and rate limiting to mitigate DDoS threats.

**Benefits:**

- Advanced threat protection for APIs and web applications.
- Blocks malicious traffic before it reaches the backend.
- Reduces security vulnerabilities and compliance risks.

## Lambda@Edge – Security Headers Enforcement

AWS Lambda@Edge is used to execute custom logic at CloudFront edge locations, allowing security headers to be applied before responses are sent to users. This ensures strict security policies and prevents common web security vulnerabilities.

### Requirements:

- A Lambda@Edge function that modifies HTTP headers to include security policies.
- Integration with CloudFront to apply headers globally.
- Custom rules for CORS, Content Security Policy (CSP), and XSS protection.

### Benefits:

- It prevents security threats like XSS, Clickjacking, and CSRF attacks.
- Improves compliance with security standards (e.g., OWASP, GDPR, HIPAA).
- Reduces load on the backend by handling security at the edge.

## AWS Certificate Manager (ACM) – SSL/TLS Security

AWS ACM provides SSL/TLS certificates to encrypt communications between the user and the application, ensuring secure HTTPS connections.

### Requirements:

- SSL/TLS certificate for both the frontend and backend domains.
- Automatic renewal to prevent certificate expiry issues.
- Integration with ALB and CloudFront.

### Benefits:

- Ensures encrypted communication between users and the system.
- Simplifies certificate management with autorenewals.
- Improves security compliance (GDPR, HIPAA).

## Amazon EC2 – Backend Compute Instance

Amazon EC2 provides a virtual machine to host the backend Node.js API, running on an Nginx proxy server for handling requests efficiently.

### Requirements:

- EC2 instance running Ubuntu with Node.js, Nginx, and PM2 for process management.
- Security group rules to allow only ALB traffic.
- Autoscaling setup for handling increased API load.



**Benefits:**

- High flexibility to scale compute power as needed.
- Customisable networking and security configurations.
- Reliable API hosting with minimal downtime.

## NGINX – Reverse Proxy for backend

Nginx is used as a reverse proxy server to handle incoming requests to the Node.js API running on EC2.

**Requirements:**

- Nginx configuration to route requests to Node.js.
- SSL termination using certificates from ACM.
- Load balancing capabilities.

**Benefits:**

- Enhances backend performance by handling concurrent connections efficiently.
- Improves security by hiding the internal structure of the backend.
- Facilitates SSL termination to manage encrypted connections.

## Application Load Balancer (ALB) – Backend Load Balancing

AWS ALB distributes traffic between multiple EC2 instances running the backend API, improving fault tolerance and reducing downtime.

**Requirements:**

- ALB listener rules to forward traffic from HTTPS (443) to backend instances.
- Health checks to ensure only healthy instances receive traffic.
- Security group rules restricting access only to Nginx proxy servers.

**Benefits:**

- Ensures backend availability & redundancy.
- Balances traffic efficiently to prevent overloading a single instance.
- Auto scale to handle sudden traffic spikes.

### 3.7.5 Development Environment – Security, Code Quality, Testing

The development environment for this project aims to ensure code quality, implement security, and implement best practices all through the software development lifecycle. The development pipeline has included several tools and techniques to do this. These technologies improve all code linting, formatting, commit validation, security scanning, and automated testing.

#### Code Quality & Linting

The following technologies help implement best practices and avoid code smells:

**ESLint:**

- A static analysis tool that helps find syntax problems, enforce code styles, and prevent possible bugs
- Best practices in JavaScript and TypeScript.

**Prettier:**

- Automatically formats code for readability and consistency.
- Maintain a uniform coding style across different team members.
- Integrated with ESLint to avoid conflicts between linting and formatting.

**Husky:**

- Enforces Git hooks to prevent bad commits and run linting and tests before pushing code.
- Follow defined coding standards before committing changes.
- Integrated with ESLint and Prettier to block commits with syntax/style violations.

**Commitlint:**

- Enforces a structured commit message format following Conventional Commits.
- Improves commit history readability and helps with automated versioning and changelogs.

## Security & Vulnerability Scanning

In a healthcare related system, security is of the greatest concern as it concerns private patient information. The following tools are combined to find security weaknesses and stop exploits:

**Snyk:**

- Scans dependencies for security vulnerabilities and suggests fixes.
- Provides Realtime alerts if a thirdparty package contains security risks.
- Help automate dependency management to prevent outdated or insecure packages from being used.

**SonarQube:**

- Analyses code for bugs, vulnerabilities, and security risks.
- OWASP security best practices for secure development.
- Provides detailed insights into code quality metrics, such as maintainability, reliability, and security.

**Helmet.js (for Node.js API):**

- Adds secure HTTP headers to protect against common web security threats such as XSS, Clickjacking, and MIME sniffing attacks.
- Used in combination with Lambda@Edge for security headers at the CloudFront level.

## Automated Testing and Quality Assurance

Testing guarantees the system works as intended and helps to avoid regression problems.

Different facets of the system are covered by several testing frameworks:

### Jest

- Unit testing of backend API functions and business logic.
- Ensures that individual functions work correctly before integration.

### Vitest

- Lightweight alternative to Jest, specifically optimised for faster test execution.
- Used in frontend components testing to validate UI logic.

### Playwright

- End to end (E2E) testing framework for simulating real user interactions.
- Used to test the React frontend and its integration with the backend API.
- Cross browser compatibility and user flow correctness.

### Insomnia

- API testing and automation to validate endpoint behavior.
- Used for manual and automated API testing before deployment.

## 3.7.6 Continuous Integration & Continuous Deployment

Using GitHub Actions, a Continuous Integration (CI) pipeline will be included to automate testing, security validation, and code quality checks. Maintaining code quality, security, and dependability of the Doctor Pharmacist Secure Prescription System depends on the use of Continuous Integration (CI) and Continuous Deployment (CD). The CI/CD pipeline guarantees that every code change is validated, secure, and effectively deployed to the production environment.

## YAML Syntax & GitHub Actions for CI/CD

To automate software development processes, GitHub Actions describes methods like building, testing, security analysis, and deployment using YAML (.yaml) configuration files. YAML is a human readable data serialisation format that is often used for configuration files given its simplicity and indentation based structure.

YAML (Yet Another Markup Language) is quite popular for designing workflows in GitHub Actions. YAML files use a key value structure and depend on whitespace indentation rather than brackets or commas.

## GitHub Actions Workflow Structure

- **Name** – Process name
- **Triggers (on)** – events activating the workflow under Triggers: push, pull request, main.

- **Jobs** – A set of actions executed in parallel or sequentially.
- **Steps** – Individual tasks executed in sequence.
- **Runners** – The operating system (Linux, Windows, macOS) where jobs execute.



Figure 11 Basic Pipeline

## Backend CI/CD Pipeline Overview

Using GitHub Actions to automate testing, security checks, and deployment, the CI/CD process of the backends runs in three stages.

### **ci.yml** – Continuous Integration Pipeline

- Runs linting, formatting, and commit validation
- Executes unit and integration tests
- Ensures code quality before merging

### **security.yml** – Security & Vulnerability Scanning

- Runs Snyk for dependency vulnerability detection
- Executes SonarCloud for static code analysis
- Ensures the backend code is secure

### **deploy.yml** – Automated Deployment Pipeline

- Sets up environment variables using GitHub Secrets
- Deploys the backend to the EC2 instance
- Restarts the PM2 process to apply new changes

## CI/CD Benefits for Backend:

- **Security Compliance** – Using Snyk and SonarCloud, finds vulnerabilities before deployment.
- **Automated Deployments** – Reduces human effort by deploying straight to AWS EC2.
- **Sero Downtime Updates** – Backend is updated without service interruptions.
- **Efficient Issue Detection** – Catches bugs and security flaws early in development.

## Frontend CI/CD Pipeline Overview

The frontend CI/CD pipeline is responsible for code validation, testing, security checks, and automatic deployment to Amason S3 and CloudFront.

### **ci.yml** – Continuous Integration Pipeline

- Runs unit tests with Vitest
- Executes ESLint and Prettier for formatting and linting
- Commit message validation

### **playwright.yml** – EndtoEnd Testing Pipeline

- Runs Playwright for UI testing
- Uploads test reports for visibility
- Caching to improve test efficiency

### **security.yml** – Security & Vulnerability Scanning

- Runs Snyk to detect dependency vulnerabilities
- Executes SonarCloud for code security analysis

### **deploy.yml** – Automated Deployment Pipeline

- Configures AWS credentials for deployment
- Build the React app using Vite
- Deploys to Amason S3 (development or production bucket based on branch)
- Creates a CloudFront cache invalidation for real time updates

## CI/CD Benefits for Frontend

- **Ensures Code Quality** – Linting and formatting guarantee clean, readable code.
- **Enhances Security** – Vulnerability scanning protects against security flaws.
- **Automates Testing** – Unit tests (Vitest) and E2E tests (Playwright) prevent regressions.

- **Fast Deployment** – AWS S3 and CloudFront automation ensure smooth releases.
- **Optimised Performance** – CloudFront invalidation ensures users get the latest UI instantly.

## 3.8 Conclusion

The system will provide an easy user interface, prescription management, role based access control, along with secure authentication. Technical criteria are cloud architecture, security policies, and automated approaches for data protection laws compliance and scalability.

## 4 Design

### 4.1 Introduction

The Secure prescription systems design phase began after completing the requirements phase. This section covers technical and visual aspects, with a focus on system structure, ux design, and smooth integration.

### 4.2 System Architecture

As shown in the figure 12 below, the architecture of this system is a three tier client server architecture where the client (doctor or pharmacist) sends HTTP requests to the server (NodeJS with express), which in turn communicates with the database (MongoDB).

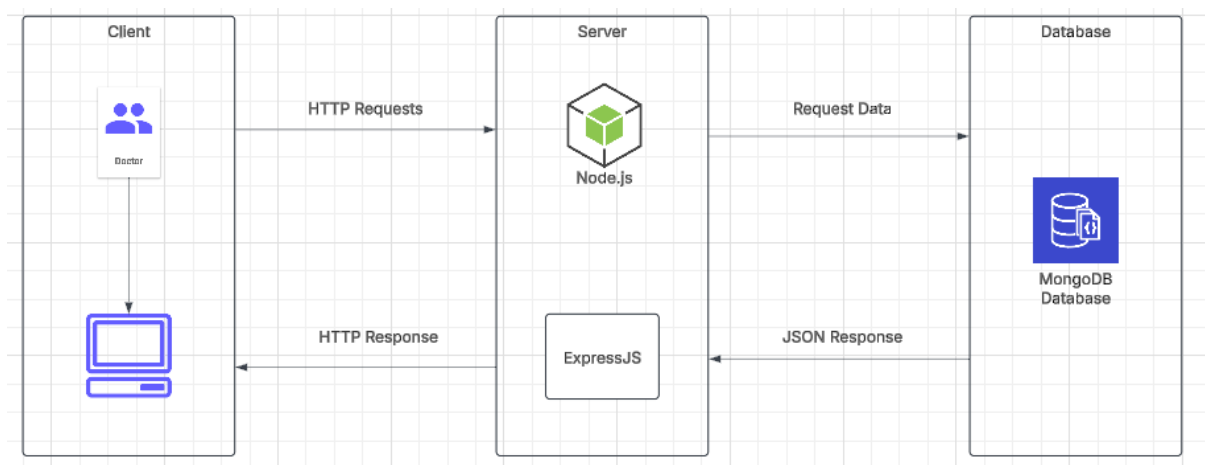


Figure 12 System Architecture

- **Client Layer (Frontend):** The client represents the doctor or pharmacist accessing the system through a web interface that is developed using React.js alongside TypeScript. The client communicates with the server by sending HTTP requests (via Axios/Fetch API).
- **Server Layer (Backend):** The server built using NodeJS and express.js handles authentication, authorisation, prescription processing, pharmacist assignments, and adding patients. It manages the business logic, notifications, and API endpoints. Realtime notifications are created through socket.io. The server then validates HTTP requests, authenticates users, processes them, and will interact with the database, that returns a JSON response with requested data.

- **Database Layer (MongoDB):** MongoDB is a NoSQL database used for storing users, maintaining encrypted medical records, prescription data, and medications. It responds with a JSON response when a query is made to the database.

## 4.3 Application Design

### 4.3.1 Technologies

#### 4.3.1.2 Frontend Technologies

- **React:** React is a JavaScript library, that allows you to use modular and reusable UI components, while working with TypeScript for improved type safety, runtime error reduction, tooling support, bug reduction, maintainability, and code reliability and collaboration.
- **Vite:** Vite is a build tool for fast development builds and optimised production output, significantly faster than webpack due to its ES module based hot module replacement and optimised tree shaking and code splitting.



Figure 13 Frontend Tech

- **ShadCN:** is used for modern, accessible, and highly customisable UI components. Shadcn provided prestyled components with support for dark and accessibility. It also integrates easily with Tailwind CSS and makes sure of a consistent UI design across the application.
- **TanStack Query:** TanStack Query is a tool for data fetching, caching, and synchronisation with the backend API, it reduces unnecessary calls and optimising performance as the application grows.



- **TanStack Router:** TanStack router manages client side navigation with typesafe route management, nested routing, code splitting, and optimisation for TanStack Query, offering parallel Route Loaders, automatic prefetching, and error boundaries.
- **Figma:** Creating interactive UI prototypes and user interface designs. Figma provided design specs for react component implementation.
- **Sod:** Sod is a TypeScriptfirst schema validation package that ensures data accuracy and type safety in form data and API POST requests, error messages and interaction with form libraries like React Hook Form.
- **ReactHookForm:** React Hook Form is a package that manages form state and validation, realtime validation feedback, and lightweight compatibility with schema validators like Sod.
- **Socket.io Client:** The Socket.IO client library enables realtime communication between the frontend and backend, enabling pharmacists to receive prescription notifications from doctors, enhancing responsiveness and interaction through persistent WebSocket connections.
- **Tailwind CSS:** This CSS framework, which prioritises functionality, is used to style frontend applications. Through the direct use of predefined classes in the HTML framework, it facilitates quick user interface development with responsive and uniform design. Tailwind facilitates accessibility, dark mode, and custom theming while integrating easily with ShadCN components.

#### 4.3.1.2.1 *Backend Technologies*

- **NodeJS:** Node.js is a JavaScript nonblocking runtime environment, built on the V8 engine, enabling developers to create servers, web applications, command line tools, and scripts, with asynchronous, event driven architecture for high load performance.
- **ExpressJS:** Is a fast, lightweight web framework for handling API requests. It also supports middleware for authentication, logging and error handling. Simplifies REST API creation for handling prescriptions, users, and orders.
- **Morgan:** Morgan is a request logging middleware that records incoming HTTP requests, helping developers track API interactions, monitor performance, and debug issues.
- **JWTs:** JWT is a secure authentication mechanism used to generate signed tokens for user sessions. It enables stateless authentication, reducing server side session management overhead while ensuring secure API access.

- **CORS:** Cross Origin Resource Sharing (CORS) is a security feature that allows controlled access to APIs from different origins. It ensures that only trusted frontend applications can communicate with the backend, preventing cross origin security vulnerabilities.
- **Nodemon:** Nodemon is a development tool that automatically restarts the Node.js application whenever changes are detected. This streamlines development by eliminating the need to manually restart the server after making code updates.
- **OTPLib:** OTPLib is a library used to generate and verify Time based One Time Passwords (TOTP) for Multifactor Authentication (MFA), enhancing account security.
- **Dotenv:** Dotenv is used to load environment variables from a .env file into the application. This ensures sensitive configuration details, such as API keys and database credentials, are not hardcoded in the codebase.
- **Cookieparser:** CookieParser is an Express.js middleware that parses incoming HTTP cookies, enabling session based authentication and user session management.
- **Bcrypt:** is a cryptographic hashing algorithm used to securely hash user passwords before storing them in the database. It implements salting to prevent brute force attacks.
- **Node:crypto module:** The built in crypto module in Node.js provides encryption, hashing, and random token generation, ensuring secure handling of sensitive data.
- **Expressvalidator:** ExpressValidator is a middleware that validates and sanitises incoming API requests. It prevents malicious inputs, enforcing data integrity and security.
- **Helmet:** Helmet is an Express.js middleware that enhances API security by configuring secure HTTP headers. It protects against vulnerabilities like cross site scripting (XSS) and clickjacking.
- **Nodemailer:** Nodemailer is a Node.js module that enables email sending directly from the backend server, supporting SMTP and OAuth2 transport protocols for HTML formatted emails with attachments or dynamic content.
- **xml2js:** This Node.js lightweight XML parser. JavaScript objects are created from XML data using JS. Structured XML input may be parsed into JSON format so that the program may process it using xml2js. This facilitates the smooth integration of external data into the system and makes working with third party data types easier.

#### 4.3.1.3 Database Technologies

- **MongoDB:** MongoDB is a NoSQL database that provides flexible schema design, high scalability, and fast query performance. It supports document based storage, indexing, and replication, ensuring efficient data management.



Figure 14 MongoDB

- **Mongoose:** Mongoose is an Object Data Modelling (ODM) library for MongoDB. It provides a structured schema definition, built in data validation, and middleware support, simplifying database interactions.

#### 4.3.1.3 Cloud Technology

A variety of AWS Cloud services will be used in the systems deployment to enable scalability, performance, and security. Frontend and backend issues are separated by the cloud infrastructure, which also enhances worldwide content delivery and guarantees secure communication.

The following services will be used by the architecture:

- **Amazon Route 53** for custom domain routing
- **Amazon CloudFront** as a global content delivery network
- **Amazon S3** to host the React frontend
- **AWS Certificate Manager (ACM)** for managing SSL/TLS certificates
- **AWS Web Application Firewall (WAF)** for threat mitigation
- **AWS Lambda@Edge** for applying security headers
- **AWS EC2** to host the backend Node.js API
- **Application Load Balancer (ALB)** for backend traffic routing
- **NGINX** as a reverse proxy on the EC2 instance

- **MongoDB Atlas** for cloud based database hosting

To maintain the systems security, high availability, and performance across many geographic locations, these services were chosen.

### 4.3.2 Design Patterns

Software design patterns are very important tools for developers as they provide proven solutions to common problems that are encountered during software development. If developers can apply these patterns to their own project, they can create more robust, maintainable and scalable software systems.

Design patterns act as reusable solutions for typical software design challenges. Design patterns provide a standard terminology and are specific to scenarios and problems. They are not finished code but templates or blueprints. (Geeksforgeeks, 2017)

### Key Characteristics of Design Patterns:

- **Reusability:** The patterns can be applied to various projects and problems, saving time and effort.
- **Standardisation:** Provide a shared language and understanding among developers
- **Efficiency:** Developers can avoid finding the solution to the same recurring problems, which leads to faster development
- **Flexibility:** Patterns are abstract solutions/templates that can be adapted to fit the requirements

Below will go into more detail around my chosen design pattern for this project

#### 4.3.2.1 MRC Design Pattern

The Model Router Controller (MRC) pattern is a variation of the classic Model view controller (MVC) pattern, specifically tailored for building RESTful APIs as there is no direct UI layer in this architecture. In this pattern, the view is replaced by the router because the API will server JSON responses instead of rendering HTML. The MRC pattern helps maintain a structured, modular, and scalable backend by separating concerns into three layers:

### Model (M) – Data Layer

The model represents the data structure and is responsible for interacting with the database. In this case, MongoDB will be used alongside Mongoose as the ODM to define schemas and interact with the database efficiently.

Responsibilities:

- Define Schema and data validation

- Implement data relationships and indexing
- Provide querying mechanisms (CRUD operations).
- Encrypt or sanitise sensitive data before storing.

## Router (R) – Routing Layer

The router acts as the middle layer between client requests and corresponding controller functions. It defines the API Endpoints and maps them to the right controller function.

### Responsibilities:

- Define RESTful HTTP Endpoints (GET, POST, PUT, DELETE)
- Route requests to the appropriate controller function.
- Apply middleware functions (e.g., Authentication, role based access control).
- Ensure modularity and maintainability of the API.

## Controller (C) – Business Logic Layer

The controllers need to contain the core logic of the API. It processes the requests, interacts with the Model, and returns responses.

### Responsibilities:

- Implement Business logic (e.g., Data validation, prescription assignment)
- Interact with the Models to perform CRUD Operations.
- Handle errors and validation responses
- Ensure data is formatted properly before sending it back

### 4.3.2.2 REST API

An API is a secure interface used by computer systems to exchange data over the internet. It defines rules for communication and acts as a gateway between clients and web resources. Representational State Transfer (REST) is software architecture that imposes conditions on APIs, enabling high performing, reliable communication on complex networks. A uniform interface is crucial for RESTful webservice design, indicating that server information is transferred in a standard format, known as a representation.

In REST architecture, stateless refers to a communication method in which the server completes every client request independently of all previous requests. Clients can request resources in any order, and every request is stateless or isolated from other requests. (AWS, 2024)

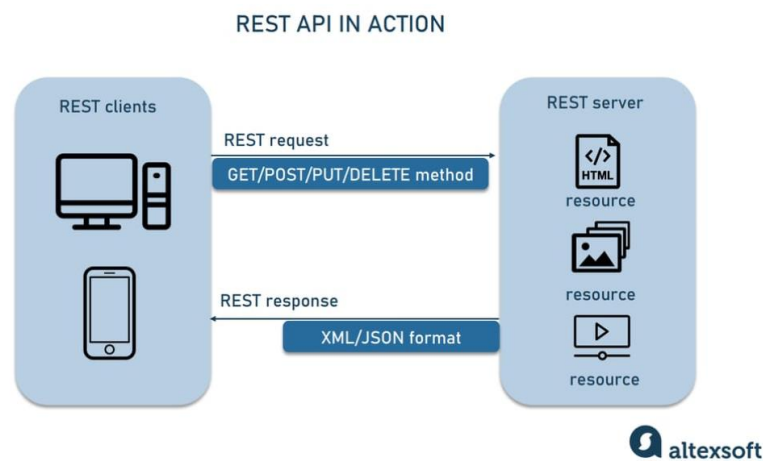


Figure 15 REST API

### 4.3.3 Database Design

#### 4.3.3.1 Introduction to Database Design

Database Design is a crucial aspect of developing a secure, scalable, and high performing application. The database structure determines how the data is stored, retrieved, and managed, impacting the efficiency and security of the system. In this project, MongoDB Atlas will be used as a NoSQL database, designed to store structured and semi structured data effectively.

### Why MongoDB?

MongoDB is a document-oriented NoSQL database that provides:

- **Scalability:** Can handle large volumes of data using sharding and replication.
- **Flexibility:** Supports dynamic Schema to allow changes without breaking the database.
- **High Performance:** Indexing and embedded documents to improve query speed
- **Security:** Supports encryption, access control, and IP whitelisting

#### 4.3.3.2 Database Schema Overview

The database is designed using a relational structure within a NoSQL framework. It consists of the following main collections (tables in SQL terms):

- Users – Stores doctors and pharmacists
- Patients – Stores patient records
- Prescriptions – Stores medical prescriptions
- Medications – contains real distributable medications in Ireland
- Appointments – Contains the patient appointments

Each collection uses references (ObjectIds) to efficiently manage relationships

#### 4.3.3.3 Entity Relationship Design

An ERD diagram is a type of flowchart that illustrates how “entities” such as people, objects or concepts relate to one another. ER diagrams are often used to design or debug relational databases. Also known as ERDs, they use a defined set of symbols, such as rectangles, diamonds and ovals. (LucidChart, 2024)

ER diagrams are related to data structure diagrams (DSDs), which focus on the relationships of elements within entities instead of the relationships between the entities themselves.

##### Uses:

- **Database design:** Used to model and design relational databases. In terms of logic and business rules and in terms of specific technology to be implemented.
- **Database troubleshooting:** ER diagrams are used to analyse existing databases to find and resolve problems in logic or deployment.
- **Guiding Implementation:** ERDs can act as a blueprint for actual database schemas. Developers can use this diagram to implement tables/models, fields, and relationships in the database.

Below in figure 16 is the ERD designed to outline the relationship between the various models.

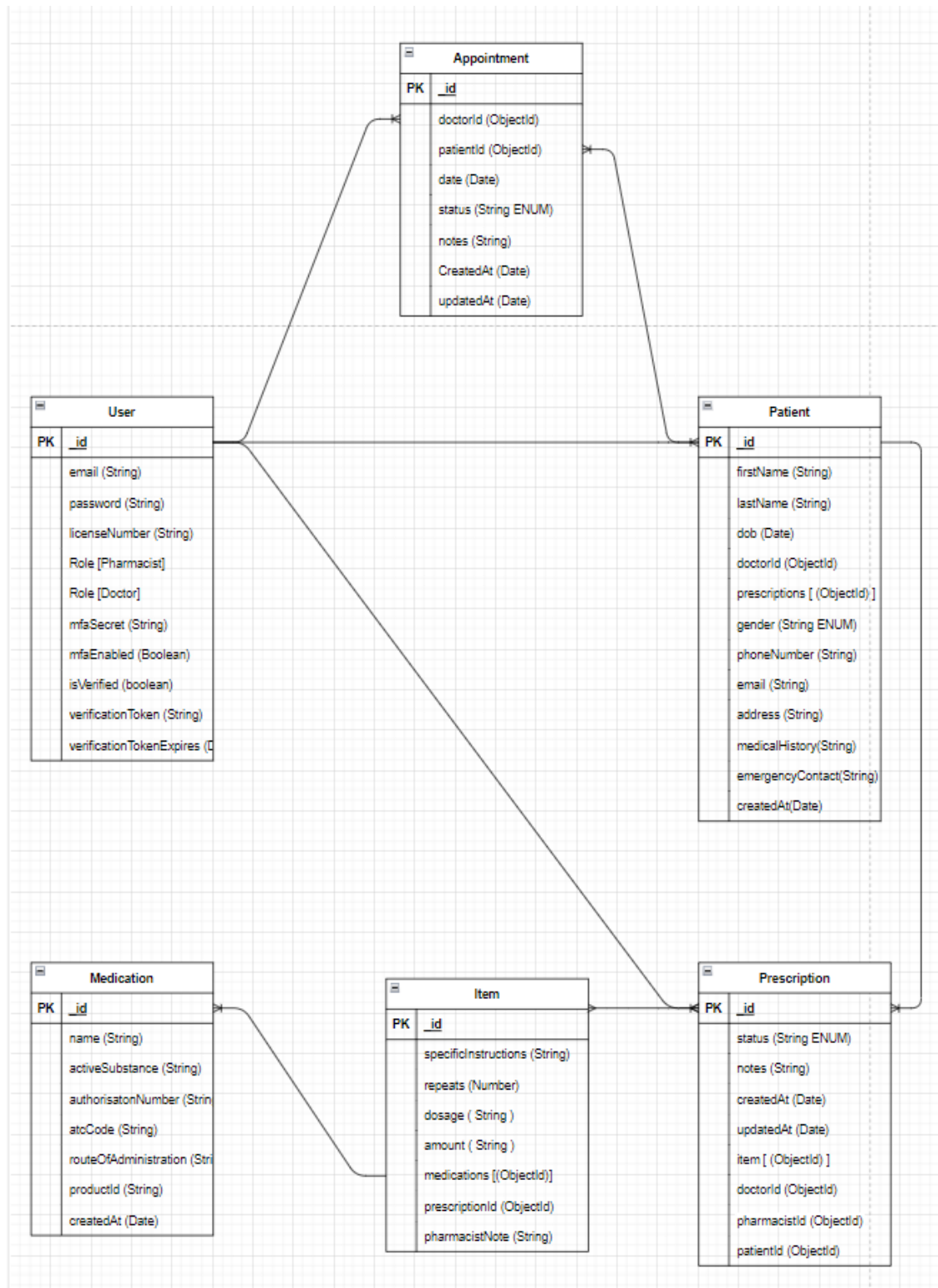


Figure 16 ERD



### 4.3.4 Process Design

Process design is a systematic method in software development that outlines the functionality and maintenance of software applications. It involves simulating data flow, input output conversion, and system component interactions. The main objective is to ensure the software operates effectively and predictably, using tools like flowcharts, sequence diagrams, and activity diagrams to visualise and record system behaviour and logic (Sommerville, 2016).

#### 4.3.4.1 Sequence Diagrams

Sequence diagrams are a crucial part of Unified Modelling Language (UML) that visualise object interactions in a sequential order. They help model dynamic behaviour in systems, understand use cases, design system architecture, and document complex processes. They clarify system logic, define component responsibilities, and ensure system functionality. They also help developers identify workflow issues early and serve as documentation for handling complex processes (GeeksforGeeks, 2023). Below in figure 17 is the sequence diagram created for the authentication flow.

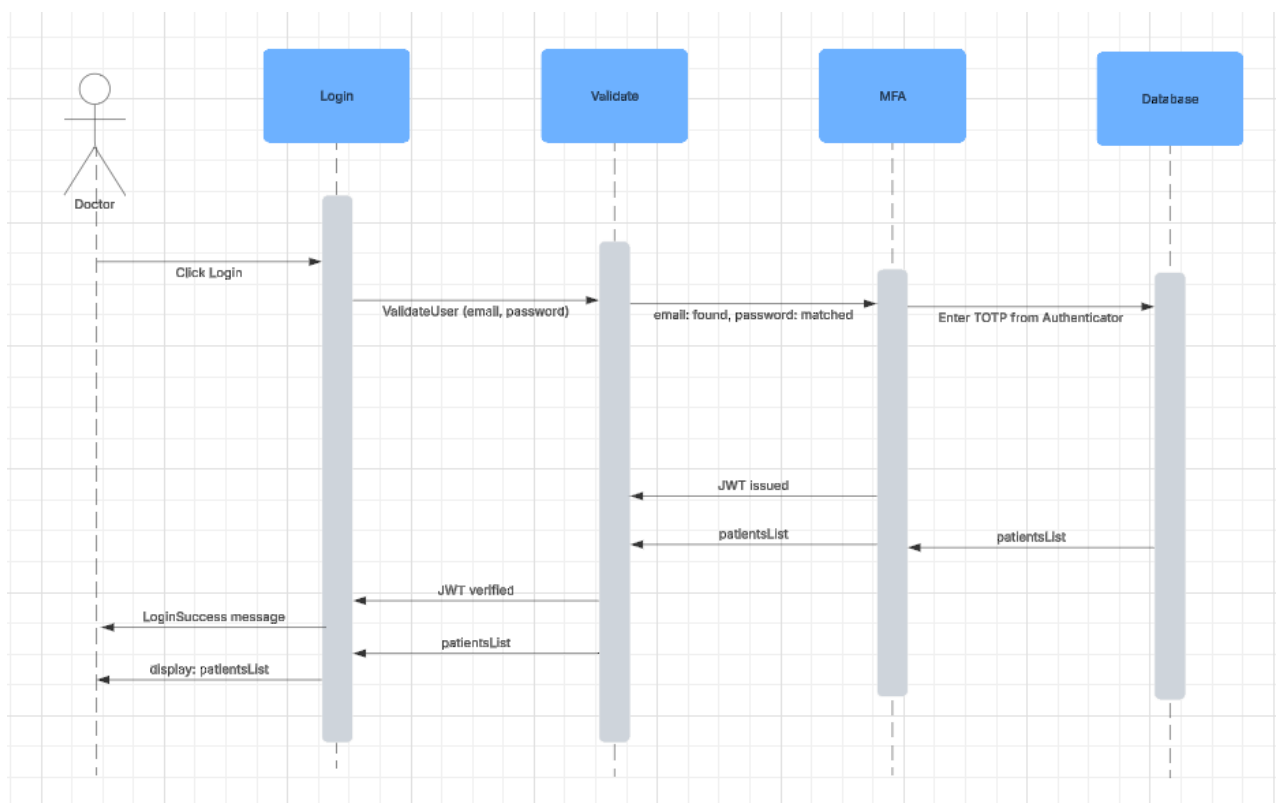


Figure 17 Sequence Diagram 1

Below in figure 18, created this sequence diagram to outline the sequence of events for a pharmacist working with the prescriptions

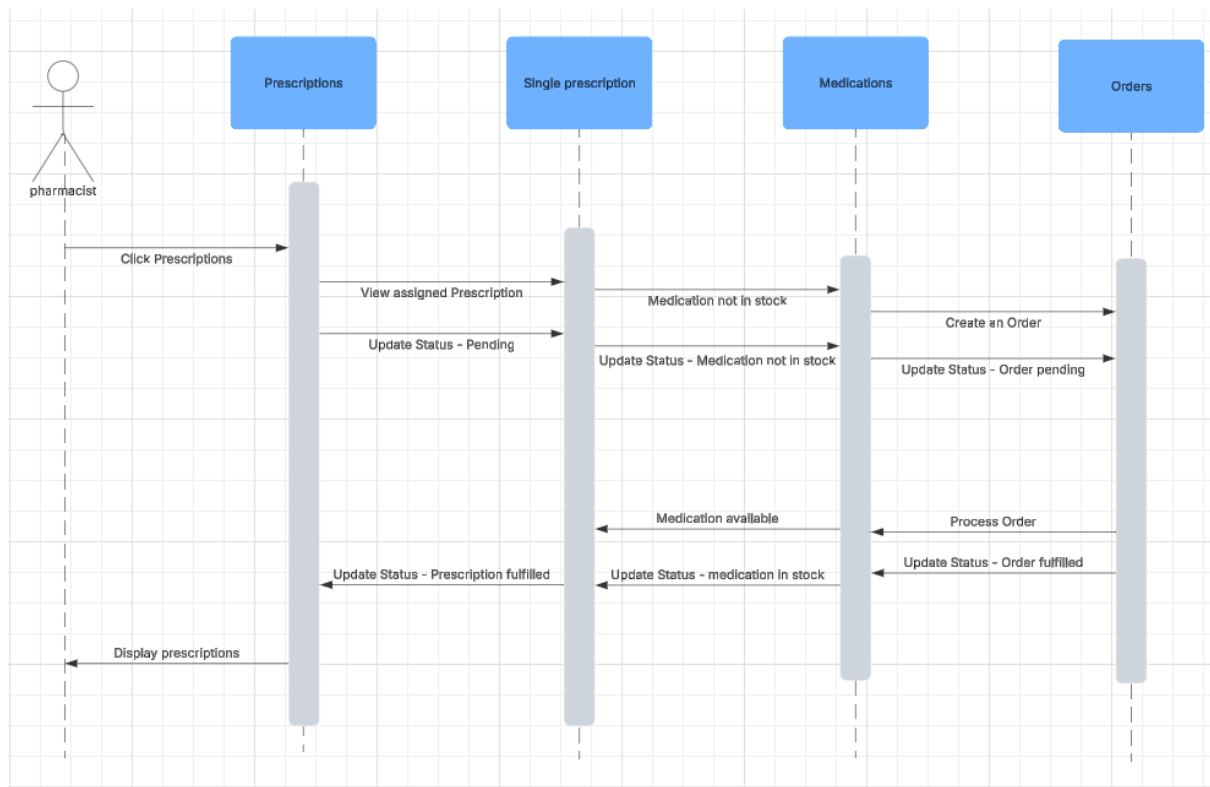


Figure 18 Sequence Diagram 2

#### 4.3.4.2 Flow Charts

A flow chart is a visual representation of a process sequence of steps and decisions, illustrating the operating processes through basic shapes and symbols. Originating from industrial engineers, flow charts are used in various fields like engineering, education, and science. They are often used in early development stages for requirement analysis, process design, system documentation, and debugging. Visualising the process flow can improve communication, minimise miscommunication, and agree on desired results, making it adaptable for both developers and non-developers (Lucidchart, 2023).

Below in figure 19, created a flow chart in Lucid chart to show the user flow for a doctor working with patients.

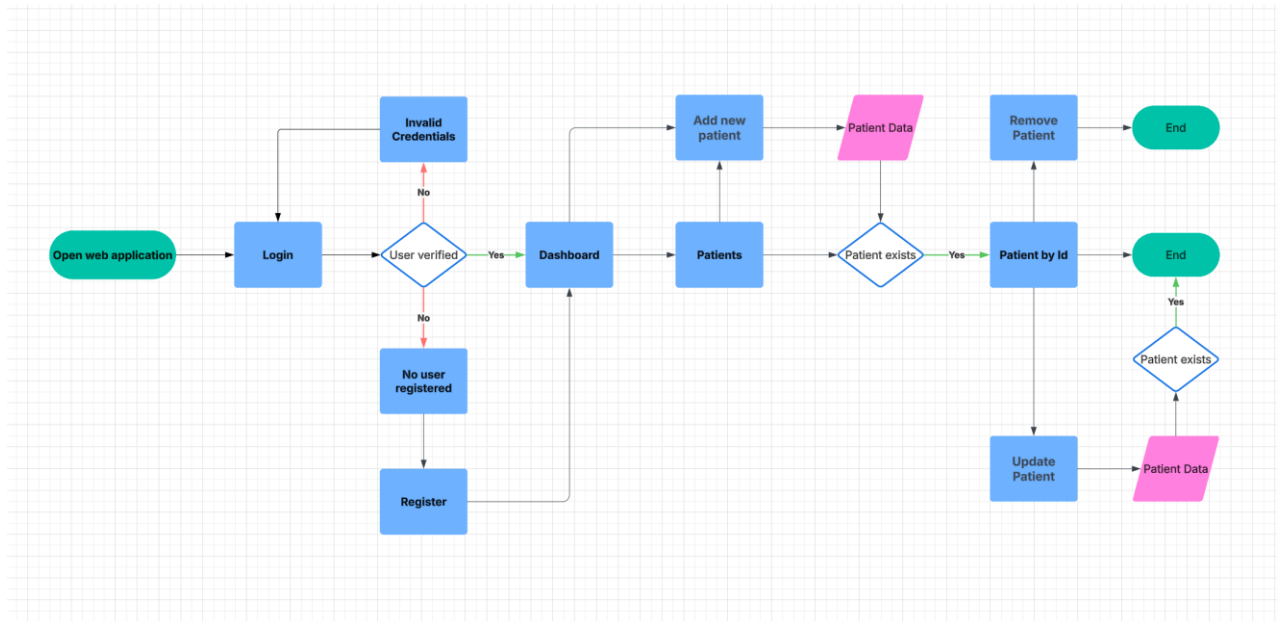


Figure 19 Flow Chart 1

In figure 20, we can see the user flow for a pharmacist working with prescription.

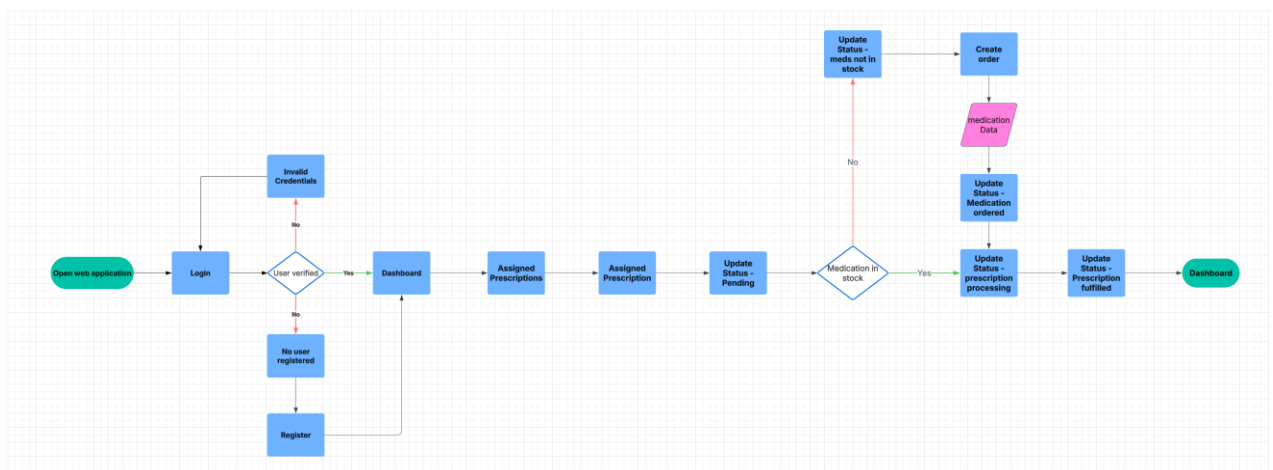


Figure 20 Flow chart 2

## 4.4 User Interface Design

The application design stage involves user interface design, using wireframes for each screen and components from the Open Source project Shadcn. Shadcn offers modern design principles, flexibility with TailwindCSS, and seamless integration with React. It provides accessible, customisable components built on Radix UI and Tailwind CSS, ensuring compliance with WCAG standards and making the application usable for a broader audience.

### 4.4.1 Wireframes

For the authentication wireframes, to present a clean and well structured user flow for logging in and registering. Here's a breakdown of the design

- Account Creation – Users enter their email, password, license number and select a role to sign up
- Multifactor Authentication (MFA) Setup – A QR Code is presented for users to scan with an authenticator app of their choice such as Microsoft authenticator.
- Login Screen – Users enter their email and password to sign in.
- MFA Code Entry – Users input the time based OTP (One Time Password) to complete the login process

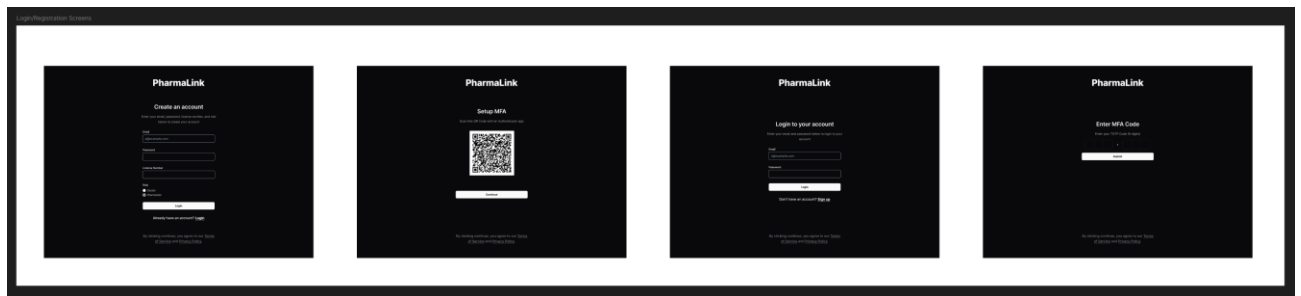


Figure 21 Authentication Designs

The main dashboard that is used for pharmacists and doctors, follows a dark theme with a data driven UI, showcasing critical patient, prescription or medical information in an organised and accessible manner. It facilitates quick navigation and decision making

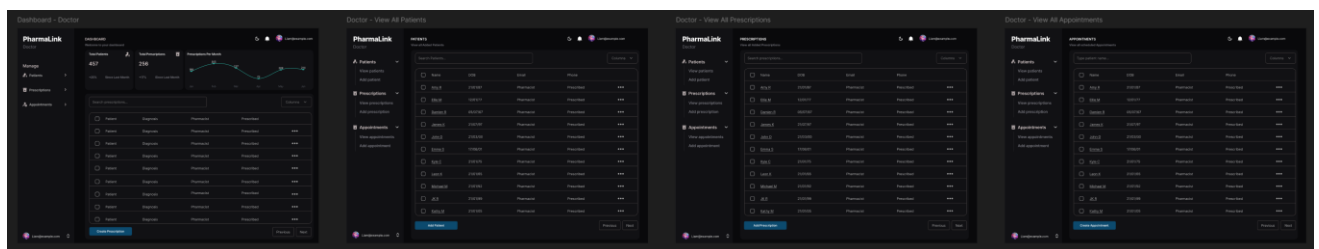


Figure 22 Dashboard designs

## 4.4.2 Design System

### 4.4.2.1 Key Components

- Typography
  - Features a clear, readable typographic hierarchy, with distinct H1, H2, H3, and paragraph styles
  - Chosen font is modern and minimalistic, ensuring readability.
  - Strong emphasis on contrast for accessibility
- Color Palette
  - Organised shades of grey, blue, and red, allowing flexibility.
  - The contrast in colours follow WCAG accessibility guidelines
- UI Components

- A collection of essential ShadCN UI elements is included to ensure consistency:
  - **Buttons** (Primary, Secondary, Destructive, Loading States)
  - **Radio Groups** (Used for selections with clear states)
  - **Text areas & Inputs** (Form elements with proper spacing and labels)
  - **Dropdowns & Select Menus** (Ensuring easy navigation and interactions)
  - **Tables & Data Lists** (For organising large data sets efficiently)
- Navigation & Sidebar
  - Dark mode optimised navigation panel for ease of access.
  - Streamlined UI for improved efficiency, reducing cognitive load.

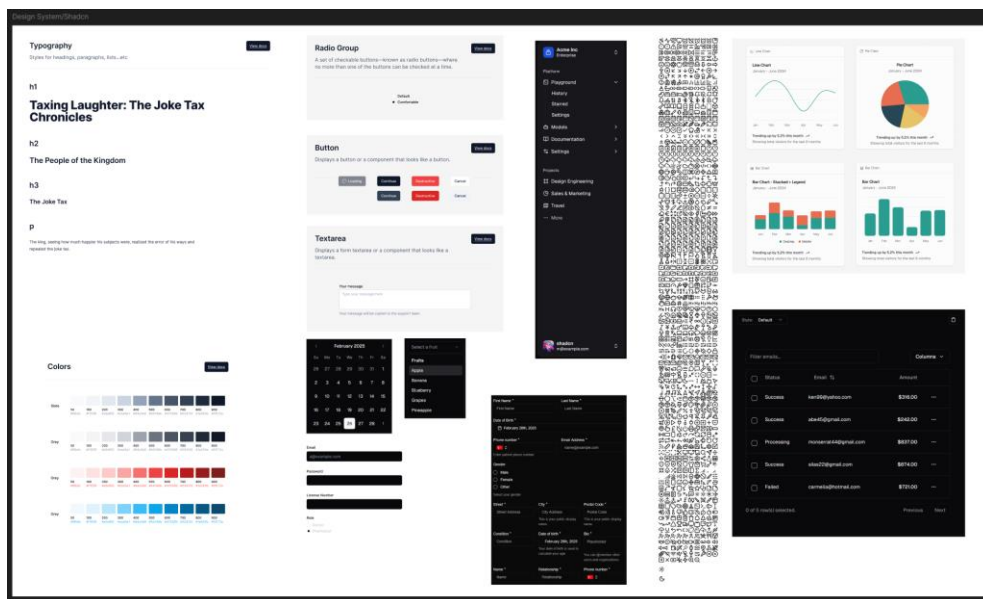


Figure 23 Design system

## 4.5 Conclusion

The PharmaLink design system is designed for doctors and pharmacists to balance usability, accessibility, and aesthetics. It uses Tailwind CSS, ShadCN, and a well organised component library for scalability and ease of development. Key design features include dashboards, structured navigation, and a dark themed interface. It supports seamless workflow for prescriptions, patient records, and appointments.

## 5 Implementation

### 5.1 Introduction

The chapter details the implementation of the Secure Prescription System, focusing on translating database schema, user interface designs, and architectural designs into a functional application. It covers the development environment, frontend and backend projects, cloud infrastructure integration, and CI/CD pipelines. The focus is on transforming conceptual models into actual code, ensuring a secure, scalable, and maintainable application.

### 5.2 Development Environment

The Secure Prescription System project was developed using a modern, developer friendly environment, including Visual Studio Code (VS Code), ESLint, Prettier, and VS Code. Git was used for version control, ensuring code quality and avoiding conflicts. Commitlint and Husky hooks were integrated into the development pipeline for standardisation. Insomnia was the primary tool for manual REST API testing, verifying backend endpoints and ensuring the API followed expected behaviour. The project also utilised Insomnia for data forms for the frontend, supporting environment variables, authentication headers, and JSON response visualisation. Below in figure 24, we can see the various API endpoints for testing with the insomnia UI.

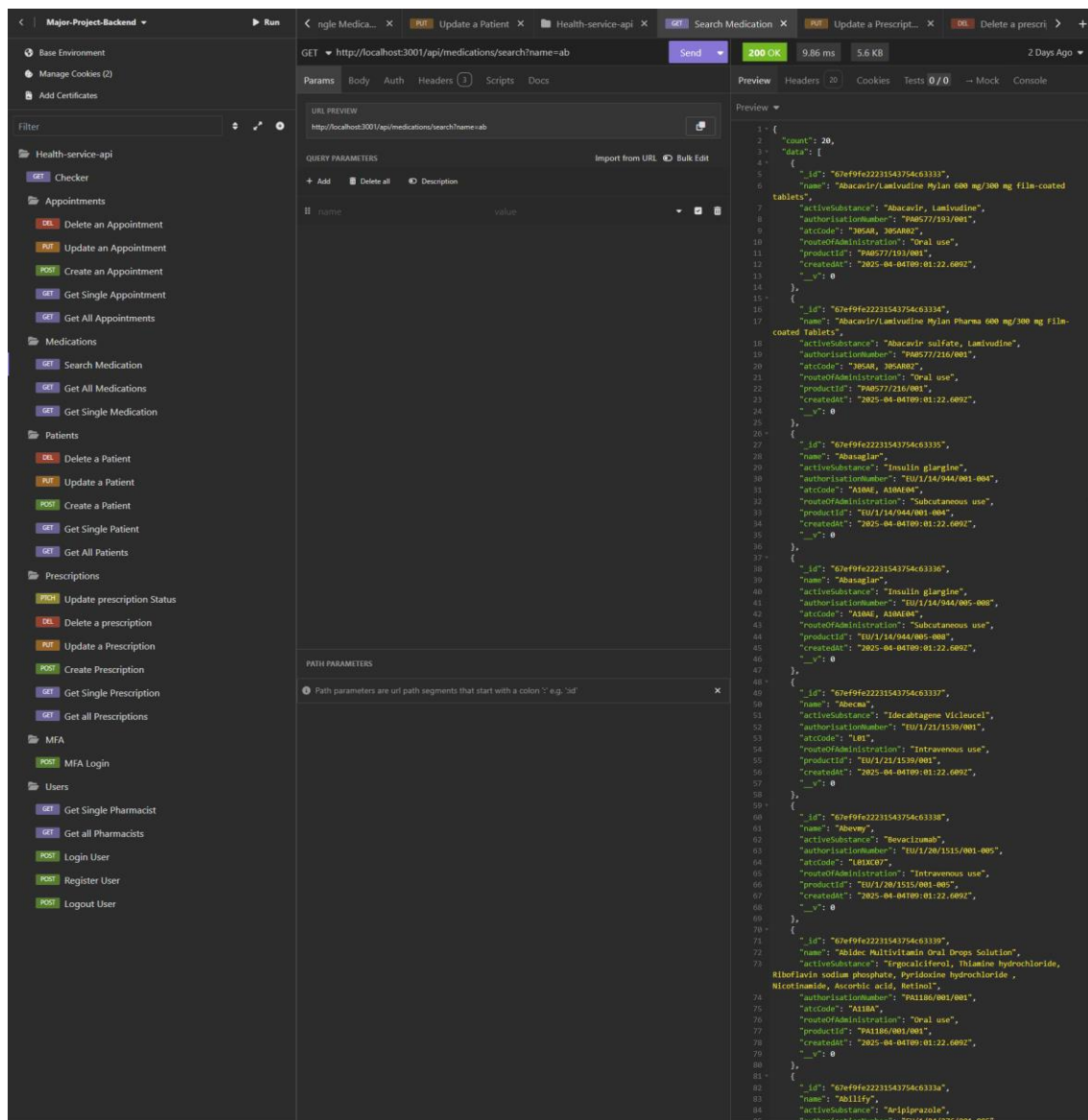


Figure 24 Insomnia

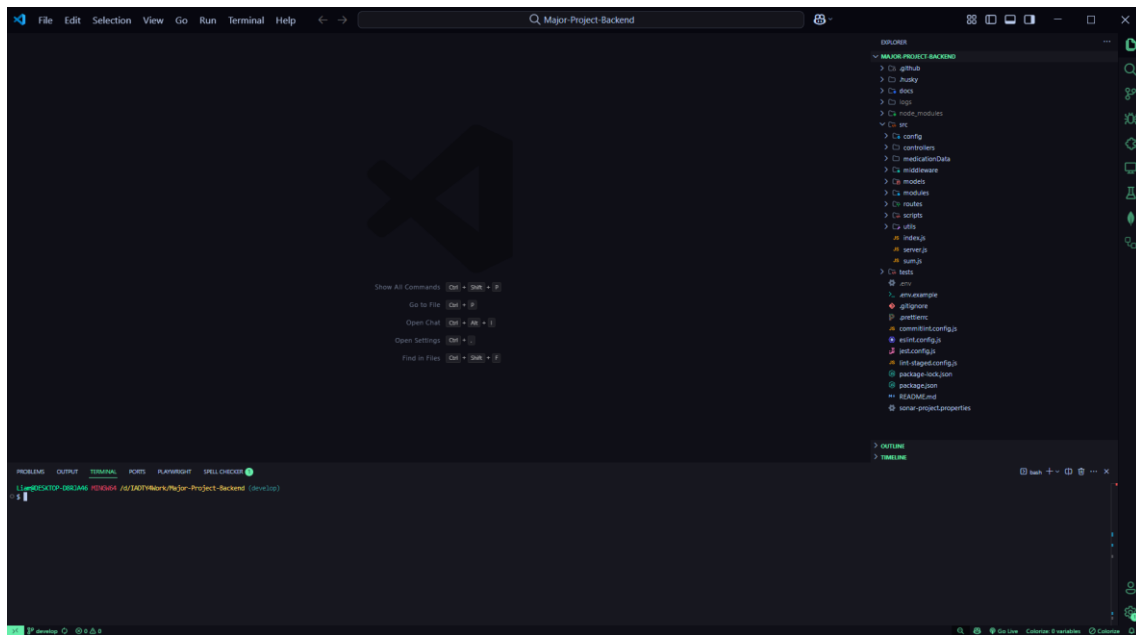


Figure 25 VS Code

## 5.3 Database

A MongoDB Atlas cluster was established for document storage, including users, patients, prescriptions, and medications. MongoDBs built in features, including IP whitelisting, monitoring, and automatic backups, ensured security and reliability. A MongoDB connection string was created from the Atlas dashboard for backend application communication, which was securely saved in a .env file under MONGO\_URI to avoid credential exposure. Below in figure 26, we can see the setting up a MongoDB cluster in MongoDB Atlas.



Connect to Major-Project

✓

Set up connection security

✓

Choose a connection method

3

Connect

### Connecting with MongoDB Driver

#### 1. Select your driver and version

We recommend installing and using the latest driver version.

Driver	Version
Mongoose	7.0 or later

Deciding between Mongoose and the Node.js Driver? [Learn more about the differences](#)

#### 2. Install your driver

**Run the following on the command line**

Node.js must be installed as a prerequisite. [Download Node.js](#)

```
npm install mongoose
```

[View MongoDB Mongoose installation instructions.](#)

#### 3. Add your connection string into your application code

**Use this connection string in your application**

☐ View full code sample

```
mongodb+srv://liamronan16:<db_password>@major-project.hzgv1.mongodb.net/?retryWrites=true&w=majority&appName=Major-Project
```

Replace `<db_password>` with the password for the `liamronan16` database user. Ensure any option params are [URL encoded](#).

RESOURCES

[Mongoose Quickstart](#)

[Access your Database Users](#)

[MongoDB & Mongoose Starter App](#)

[Troubleshoot Connections](#)

Go Back

Done

Figure 26 MongoDB Atlas

## 5.4 Cloud Infrastructure

### 5.4.1 Overview

The Secure Prescription Systems cloud infrastructure was built using AWS services and MongoDB Atlas for a scalable, secure, and high performing deployment. The infrastructure supports a three tier architecture: frontend, backend, and database. MongoDB Atlas hosts the database, while EC2 and NGINX handle backend hosting, ALB for load balancing, and Amazon Route 53 for DNS routing. Security measures include SSL/TLS and AWS WAF.

A high level architecture diagram illustrating the integration of each service may be found below in figure 27:

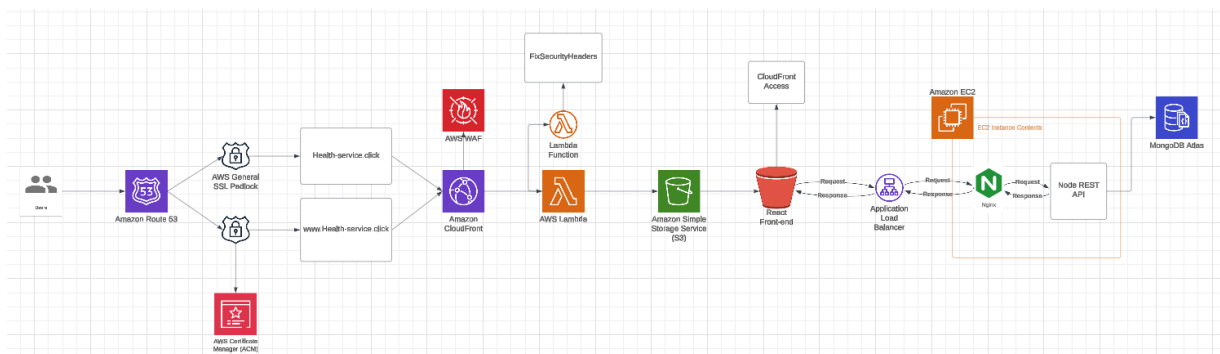


Figure 27 Cloud Infrastructure

### 5.4.2 Backend

The Secure Prescription Systems backend API was developed using Node.js and Express and deployed on an Amazon EC2 instance for scalability and control. The architecture is protected and optimised using NGINX, ALB, ACM, and DNS routing. The backend is hosted on an Ubuntu powered EC2 instance, with additional software like NGINX installed on the virtual server (Amazon Web Services, 2023).

## Amazon EC2 (Elastic Compute Cloud)

### Configuration Steps:

- Chose a t2.micro instance for the backend
- Created a new security group to only allow SSH (port 22) only from my IP, allow HTTP (port 80) and HTTPS (Port 443) access only from the ALB (application load balancer), and to block all other external access.

Security Groups (3) [Info](#)

[Actions](#) [Export security groups to CSV](#) [Create security group](#)

<input type="checkbox"/>	Name	Security group ID	Security group name	VPC ID	Description	Owner
<input type="checkbox"/>	-	<a href="#">sg-0babe3441f751cfc7</a>	ALB SG	<a href="#">vpc-0675b1128c506f6b4</a>	allows inbound from the internet on po...	730335616885
<input type="checkbox"/>	-	<a href="#">sg-03ea8f17c65b06221</a>	default	<a href="#">vpc-0675b1128c506f6b4</a>	default VPC security group	730335616885
<input type="checkbox"/>	-	<a href="#">sg-00ec9ca08fde01133</a>	launch-wizard-1	<a href="#">vpc-0675b1128c506f6b4</a>	launch-wizard-1 created 2025-01-15T2...	730335616885

Figure 28 EC2 Security Groups

- Then connected to the instance via SSH using a generated PEM key which is a container format used for storing cryptographic keys

<input type="checkbox"/>	nodeJS-api	rsa	2025/01/15 ...	60:99:3f:d0:5...	key-0acaa3823cc1a26a7
--------------------------	------------	-----	----------------	------------------	-----------------------

Figure 29 SSH PEM Key

- When inside the ubuntu machine, installed various programs and or packages such as NodeJS, PM2 for process management, NGINX to serve as a reverse proxy and git to clone the project repo.
- Pulled the backend codebase, used npm install to fetch dependencies, and created a .env file with secrets and MongoDB credentials
- Could then start the API server using pm2 start server.js and enabled boot persistence with pm2 startup

## NGINX – Reverse Proxy Setup

NGINX was configured as a reverse proxy to receive HTTP traffic from the ALB and forward it internally to the Node.js application running on the port specified in the .env file (e.g., port 3001). This adds a layer of security and abstraction while also handling tasks like compression and connection handling more efficiently than Node.js alone.

### Configuration:

May edit the NGINX config file with **sudo nano /etc/nginx/sites/available/default**

```
ubuntu@ip-172-31-23-152: ~/actions-runner-backend$ nano /etc/nginx/sites-available/default
# You should look at the following URL's in order to grasp a solid understanding
# of nginx configuration files in order to fully unleash the power of nginx.
# https://www.nginx.com/resources/wiki/start/
# https://www.nginx.com/resources/wiki/start/topics/tutorials/config_pitfalls/
# https://wiki.debian.org/nginx-directorystructure
#
# In most cases, administrators will remove this file from sites-enabled/ and
# leave it as reference inside of sites-available where it will continue to be
# updated by the nginx packaging team.
#
# This file will automatically load configuration files provided by other
# applications, such as Drupal or Wordpress. These applications will be made
# available underneath a path with that package name, such as /drupal.
#
# Please see /usr/share/doc/nginx-doc/examples/ for more detailed examples.
#
# Default server configuration
#
server {
    listen 80;
    server_name health-service.click www.health-service.click;

    # Enable Gzip compression
    gzip on;
    gzip_types text/plain application/json text/css application/javascript;
    gzip_vary on;

    add_header Content-Security-Policy "default-src 'self'; connect-src 'self' https://health-service-api.click https://health-service.click" always;
    add_header X-Content-Type-Options nosniff always;
    add_header X-Frame-Options DENY always;
    add_header X-XSS-Protection "1; mode=block" always;
    # SSL configuration
    #
    # listen 443 ssl default_server;
    # listen [::]443 ssl default_server;
    #
    # Note: You should disable gzip for SSL traffic.
    # See: https://bugs.debian.org/77332
    #
    # Read up on ssl_ciphers to ensure a secure configuration.
    # See: https://bugs.debian.org/76572
    #
    # Self signed certs generated by the ssl-cert package
    # Don't use them in a production server!
    #
    # Include snippets/snakeoil.conf
    #
    # Add index.php to the list if you are serving PHP
    index index.html index.htm index.nginx-debian.html;

    location / {
        root /var/www/frontend;
        index index.html;
        try_files $uri /index.html;
    }

    location /api {
        rewrite /api/(.*) /api/$1 break;
        proxy_pass https://health-service-api.click;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

        # CORS
        add_header Access-Control-Allow-Origin https://health-service.click always;
        add_header Access-Control-Allow-Credentials true always;
        add_header Access-Control-Allow-Methods "GET, POST, PUT, DELETE, OPTIONS, PATCH" always;
    }
}
```

Figure 30 NGINX Config

Restarted NGINX with **sudo systemctl restart nginx**. This allowed the EC2 instance to receive traffic on port 80 and forward it internally to the API via port 3001(.env), keeping the app isolated and protected.

## Application Load Balancer (ALB)

The backend API is protected and routed through an **Application Load Balancer (ALB)**, which is configured to distribute traffic securely and efficiently to the EC2 instance running the Node.js server. As shown in **Figure X**, the ALB named healthserviceapialb is set up as an **internet facing** load balancer spanning multiple availability zones for high availability.

**health-service-api-alb** Actions

**Details**

**Load balancer type**  
Application

**Scheme**  
Internet-facing

**Load balancer ARN**  
arn:aws:elasticloadbalancing:eu-west-1:730335616885:loadbalancer/app/health-service-api-alb/1cd81fe917e87113

**Status**  
Active

**Hosted zone**  
Z32O12XQLNTSW2

**VPC**  
vpc-0675b1128c506f6b4

**Availability Zones**  
subnet-0685d94cf173d184d eu-west-1b (euw1-az1)  
subnet-0b6265d13386c3e39 eu-west-1c (euw1-az2)

**DNS name**  
health-service-api-alb-1565982458.eu-west-1.elb.amazonaws.com (A Record)

**Load balancer IP address type**  
IPv4

**Date created**  
January 28, 2025, 18:31 (UTC+00:00)

Listeners and rules (2) Info Manage rules Manage listener Add listener

A listener checks for connection requests on its configured protocol and port. Traffic received by the listener is routed according to the default action and any additional rules.

Filter listeners

Protocol:Port	Default action	Rules	ARN	Security policy	Default SSL/TLS certificate	mTLS	Trust store
<a href="#">HTTP:80</a>	Redirect to HTTPS://#{host}:443/#{path}?#{query} • Status code: HTTP_301	<a href="#">1 rule</a>	ARN	Not applicable	Not applicable	Not applicable	Not applicable
<a href="#">HTTPS:443</a>	Forward to target group • <a href="#">nodejs-target-group</a> : 1 (100%) • Target group stickiness: Off	<a href="#">1 rule</a>	ARN	ELBSecurityPolicy-TLS13-1-2-...	<a href="#">health-service-api.click (Certifi...</a>	Off	Not applicable

Figure 31 Application Load Balancer

In the listeners and rules section, two key listeners are configured:

- **HTTP (port 80):** This listener automatically redirects all unsecured HTTP requests to HTTPS using a 301 redirect. This ensures that all incoming traffic is encrypted before reaching the application.
- **HTTPS (port 443):** This is the main listener that accepts secure traffic. It uses an SSL/TLS certificate issued through **AWS Certificate Manager (ACM)** for the domain healthserviceapi.click. The certificate is attached to this listener, allowing the ALB to terminate SSL connections, i.e., decrypt HTTPS traffic at the load balancer level.

The decrypted requests are sent internally to the target group, that manages the backend EC2 instance, when the SSL handshake is finished. By removing the encryption workload from the EC2 instance, this configuration increases security by guaranteeing end to end encryption for users and boosts speed.

Because the ALB is connected, harmful traffic can be filtered and blocked before it reaches the backend application. Under various traffic scenarios, the backends scalability, security, and performance are guaranteed by this multilayered setup. (Amazon Web Services, 2023).

## AWS Certificate Manager (ACM) – SSL/TLS Certificate Provisioning

Using AWS Certificate Manager (ACM), an SSL/TLS certificate was provisioned to allow users and the backend API to communicate securely over HTTPS. The Application Load Balancer (ALB) uses this certificate to manage SSL termination and was set up especially for the domain healthserviceapi.click.

The certificate has been issued, validated, and is now in use, as seen in Figure Y:

3820b256-efff-44db-9ff9-adede1518417

Delete

#### Certificate status

##### Identifier

3820b256-efff-44db-9ff9-adede1518417

##### Status

Issued

##### ARN

arn:aws:acm:eu-west-1:730335616885:certificate/3820b256-efff-44db-9ff9-adede1518417

##### Type

Amazon Issued

#### Domains (1)

Create records in Route 53

Export to CSV

Domain	Status	Renewal status	Type	CNAME name	CNAME value
health-service-api.click	Success	-	CNAME	_77e538f6bbcd788bf86eb4418c40e170.health-service-api.click.	_9d268b8validation

#### Details

##### In use

Yes

##### Serial number

0f:1d:04:de:01:78:6a:92:7b:57:1e:2b:c8:ed:a3:a3

##### Requested at

January 28, 2025, 17:39:25 (UTC)

##### Renewal eligibility

Eligible

##### Domain name

health-service-api.click

##### Public key info

RSA 2048

##### Issued at

January 28, 2025, 17:39:38 (UTC)

##### Number of additional names

0

##### Signature algorithm

SHA-256 with RSA

##### Not before

January 28, 2025, 00:00:00 (UTC)

##### Can be used with

CloudFront, Elastic Load Balancing, API Gateway and other integrated services.

##### Not after

February 26, 2026, 23:59:59 (UTC)

Figure 32 API SSL Cert

## Amazon Route 53 – Domain name management

AWS scalable Domain Name System (DNS) web service, Amazon Route 53, to handle custom domain names and redirect traffic to the Secure Prescription Systems frontend and backend endpoints. For dependable and secure domain management, Route 53's low latency and highly accessible DNS routing is crucial. (Amazon Web Services, 2023).

Route 53 to setup two custom domain names:

- Healthserviceapi.click
- Healthservice.click

#### Hosted zones (2)



View details

Edit

Delete

Create hosted zone

Automatic mode is the current search behavior optimized for best filter results. [To change modes go to settings.](#)

Filter records by property or value

< 1 > ⚙

	Hosted zone name	Type	Created by	Record count	Description	Hosted zone ID
<input type="radio"/>	<a href="#">health-service-api.click</a>	Public	Route 53	5	HostedZone created...	Z0101728CNA35TH...
<input type="radio"/>	<a href="#">health-service.click</a>	Public	Route 53	6	HostedZone created...	Z091131627CI9XN7...

Figure 33 Domain Names

Alias records were used to connect each domain to its resource, enabling interaction with CloudFront and ALB without requiring IP addresses. The ACM certificate was also validated using DNS records, allowing for safe, HTTPS based access.

A vital component of the safe, cloud native infrastructure, Route 53 guarantees low latency routing, intelligent failover, and smooth AWS integration.

### 5.4.3 Frontend

The Secure Prescription Systems React based frontend was implemented using AWS services like React with Vite, Amason S3, CloudFront, ACM, WAF, Lambda, and Route 53, ensuring fast performance, global accessibility, and robust security for a scalable and secure frontend experience under [www.healthservice.click](http://www.healthservice.click).

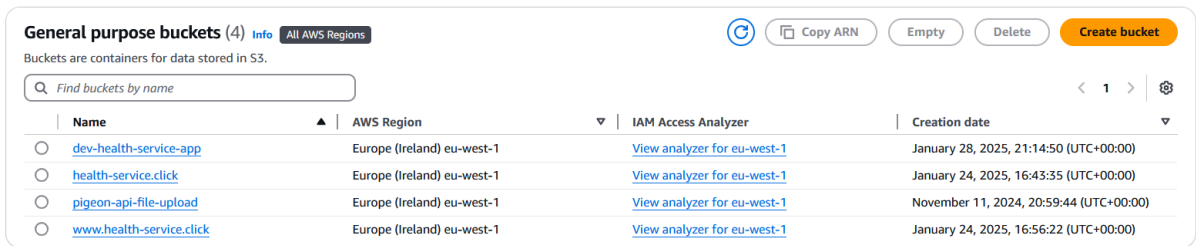
### Amason S3 – Static hosting of React App

After Vites compiling of the React application, the static files were uploaded to Amason S3, which is CloudFront origin. Two S3 buckets were built by me:

- [healthservicefrontenddev](#) for development
- [healthservice.click](#) for production

Each buckets "Properties" tab had static website hosting enabled. All public access was shut down to ensure security. Rather, set up Origin Access Control (OAC), which restricts item retrieval from the S3 bucket to CloudFront alone. This stops the files from being accessed directly from the public internet.

Used an automated workflow to deploy the application, using `npm run build` and then uploading the contents of the `dist` folder. Bucket versioning was also enabled to track deployment history and facilitate rollbacks. (Amason Web Services, 2023).



The screenshot shows the AWS S3 console interface. At the top, it says "General purpose buckets (4)" with an "Info" link and a "All AWS Regions" button. Below this, there's a search bar labeled "Find buckets by name". To the right of the search bar are buttons for "Copy ARN", "Empty", "Delete", and a prominent orange "Create bucket" button. Below the buttons is a table with four columns: "Name", "AWS Region", "IAM Access Analyzer", and "Creation date". The table lists four buckets, all in the "Europe (Ireland) eu-west-1" region. Each bucket name is a link, and each has a "View analyzer for eu-west-1" link in the IAM Access Analyzer column.

Name	AWS Region	IAM Access Analyzer	Creation date
<a href="#">dev-health-service-app</a>	Europe (Ireland) eu-west-1	<a href="#">View analyzer for eu-west-1</a>	January 28, 2025, 21:14:50 (UTC+00:00)
<a href="#">health-service.click</a>	Europe (Ireland) eu-west-1	<a href="#">View analyzer for eu-west-1</a>	January 24, 2025, 16:43:35 (UTC+00:00)
<a href="#">pigeon-api-file-upload</a>	Europe (Ireland) eu-west-1	<a href="#">View analyzer for eu-west-1</a>	November 11, 2024, 20:59:44 (UTC+00:00)
<a href="#">www.health-service.click</a>	Europe (Ireland) eu-west-1	<a href="#">View analyzer for eu-west-1</a>	January 24, 2025, 16:56:22 (UTC+00:00)

Figure 34 S3 Buckets

Below in figure 35 we can see the static files last modified date:

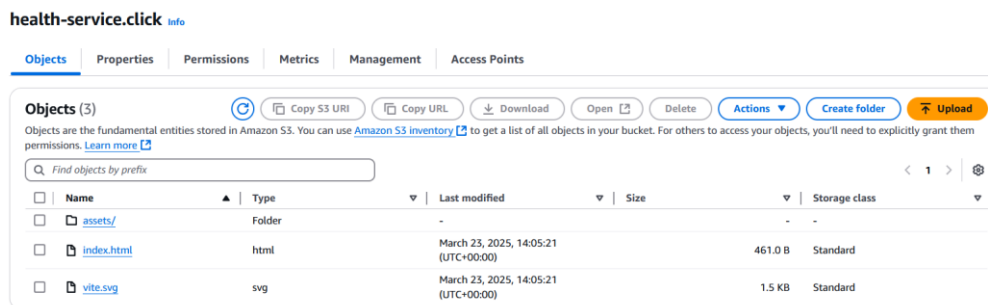


Figure 35 S3 Latest Files

## Amazon CloudFront – Global Content Delivery Network

The Amazon CloudFront distribution, with the production S3 bucket as the origin, provides global performance and minimal latency for the React frontend. CloudFront caches static assets across AWS edge locations, serving as a Content Delivery Network, reducing user access time (Amazon Web Services, 2023).

To make sure that CloudFront connection is encrypted, added an SSL/TLS certificate from AWS Certificate Manager and enforced HTTPS only access during setup. Gzip compression and caching were turned on to minimise file sizes and enhance end user load times.

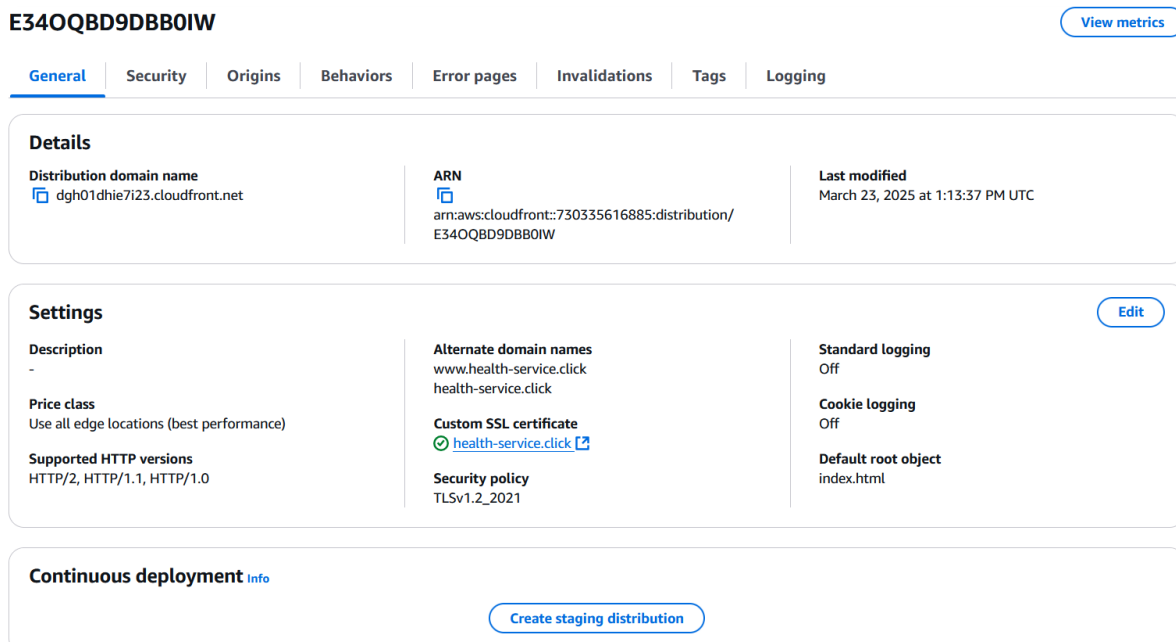


Figure 36 CloudFront Distribution



Managing cache invalidations is an essential aspect of using a CDN. CloudFront could keep serving out of date cached files from edge locations after deploying a new frontend version to the S3 bucket. Set up CloudFront invalidations to fix this. To guarantee that users receive updated files right away after deployment, sent invalidation requests, focusing on paths like `/*`. Users will always view the most recent version of the program without any delays thanks to this procedure.

Invalidations

View details

Copy to new

Create invalidation

Filter invalidations by property or value

< 1 2 3 4 > ⚙

	Invalidation ID	Status	Date created
<input type="radio"/>	<a href="#">I2E8HZKXNMHUSRJIMWMS225XTE4</a>	✔ Completed	March 23, 2025 at 2:05:21 PM UTC
<input type="radio"/>	<a href="#">I6HNCRRMMG50VQ6Y466Z7OH7N8M</a>	✔ Completed	March 23, 2025 at 1:13:43 PM UTC
<input type="radio"/>	<a href="#">IFQLLTPZU3MYFDM4X8AYGGPFQ</a>	✔ Completed	March 23, 2025 at 12:53:11 PM UTC
<input type="radio"/>	<a href="#">I1AMO17C9U6ZELWBQI9ACPPTQ8</a>	✔ Completed	March 23, 2025 at 12:44:00 PM UTC
<input type="radio"/>	<a href="#">I47BASR00R5ON3O34V36CLQNH2</a>	✔ Completed	March 23, 2025 at 12:38:21 PM UTC
<input type="radio"/>	<a href="#">I4BQEWRE82ZL8B3DS8QU39FLLY</a>	✔ Completed	March 23, 2025 at 12:33:19 PM UTC
<input type="radio"/>	<a href="#">I9E7JM1NANTOOXT6DNB12W64DL</a>	✔ Completed	March 23, 2025 at 12:21:36 PM UTC
<input type="radio"/>	<a href="#">I696HBS96Z088C47IR64XB6V64</a>	✔ Completed	March 23, 2025 at 12:14:51 PM UTC
<input type="radio"/>	<a href="#">I214DKWK1OOPB3B4FCLNZR1N6O</a>	✔ Completed	March 23, 2025 at 12:09:20 PM UTC
<input type="radio"/>	<a href="#">I927GOVL36VXZ9HFL655Z5JJE1</a>	✔ Completed	March 21, 2025 at 9:24:22 PM UTC

Figure 37 Cloud Invalidation

Security is enforced at the CDN level in addition to performance enhancement. Combined the CloudFront distribution with AWS WAF (Web Application Firewall) to stop malicious or suspicious traffic before it even gets to the S3 origin.

## AWS WAF Web Application Firewall

The CloudFront distribution and AWS Web Application Firewall (WAF) were combined to protect the frontend from common web threats like SQL injection, XSS, and bot attacks. Custom rate based rules and AWS Managed Rules were implemented to restrict excessive requests from a single IP address, reducing server burden and potential risk (Amason Web Services, 2023).

▼ Security - Web Application Firewall (WAF) Info

Keep your application secure from the most common web threats and security vulnerabilities using AWS WAF. Blocked requests are stopped before they reach your web servers.

Core protections

Enabled

► CloudFront geographic restrictions

Figure 38 CloudFront Web Application Firewall

## Lambda@Edge – Injecting Security Headers

Implemented a Lambda@Edge function that inserts HTTP security headers into each response to enforce browser level security regulations. After CloudFront retrieves content from S3, but before sending it to the client, the function is activated during the origin response phase.

Without needing modifications to the S3 files, this method allows for centralised and uniform security enforcement across all client responses (Amazon Web Services, 2023).

To enforce safe browser behaviour without requiring changes to the static files housed in S3, the functions main goal was to dynamically insert HTTP security headers into each response.

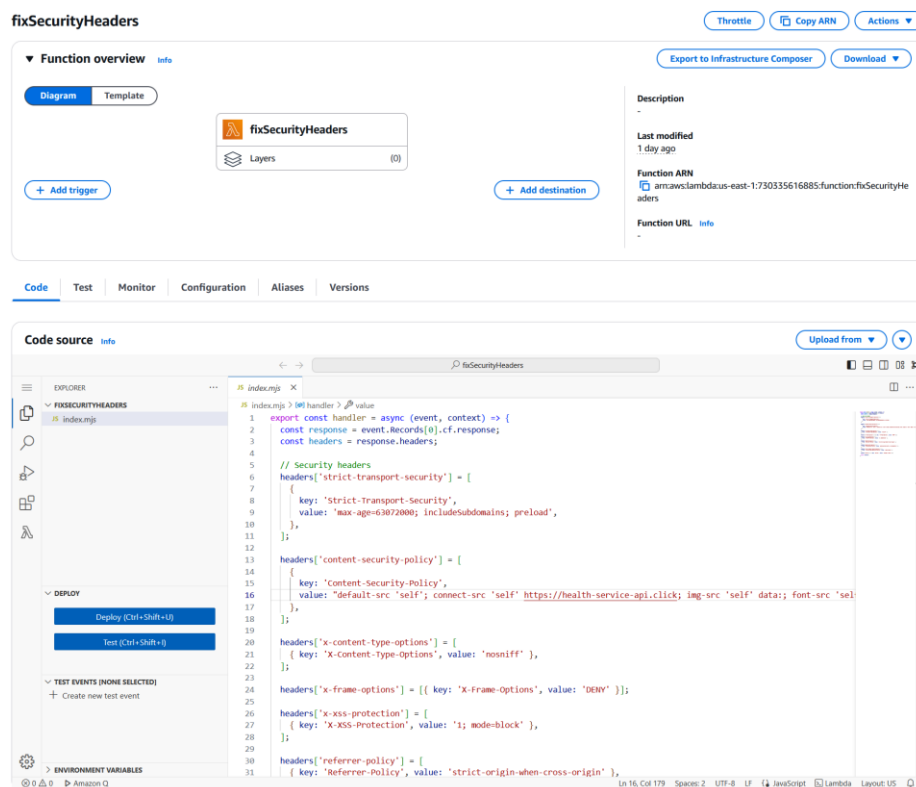


Figure 39 Lambda@edge Security Function

The function added the following headers;

- **Content Security Policy:** By specifying which sources the browser may load material from (such as scripts, fonts, and images), the material Security Policy (CSP) prevents data injection and crosssite scripting (XSS) attacks. Even if malicious scripts are introduced into the website, it stops them from running (MDN Web Docs, 2023).
- **Strict Transport Security (HSTS):** By requiring the browser to view the website exclusively over HTTPS, even if the user manually inputs http://, Strict Transport Security (HSTS) guards against protocol downgrade and man in the middle (MITM) attacks (OWASP, 2021). This guarantees always encrypted communication.

- **XContentTypeOptions:** This header stops browsers from trying to guess the file type, a practice known as MIME type sniffing. A file meant to be plain text could be interpreted and run as JavaScript if this limitation wasn't in place. XContentTypeOptions Setting: Strict MIME type compliance is enforced by no sniff (OWASP, 2021).

## AWS Certificate Manager (ACM) – SSL/TLS Encryption

Issued an SSL/TLS certificate for the domain `www.healthservice.click` using AWS Certificate Manager (ACM) to safeguard data sent between users and the application. ACM manages the automatic issue, validation, and renewal of certificates (via DNS in Route 53). After that, the certificate was added to the CloudFront distribution, allowing all client connections to be encrypted over HTTPS (Amazon Web Services, 2023).

By eliminating the need for manual certificate installation or renewal, ACM streamlines SSL administration and guarantees encrypted transmission.

### 5.4.4 Summary

The Secure Prescription System uses Amazon Web Services (AWS) for its cloud infrastructure, ensuring security, scalability, performance, and high availability. The frontend and backend follow best practices for fault tolerance, security, and deployment. An EC2 instance hosts the Node.js and Express API, while NGINX serves as a reverse proxy. An Application Load Balancer manages SSL termination with certificates provided by AWS Certificate Manager (ACM). Amazon Route 53 manages custom domain resolution, and security groups regulate access. The frontend is stored on Amazon S3 and compiled into static files using React and Vite. Amazon CloudFront distributes these files globally, and AWS WAF safeguards the distribution. ACM enforces SSL/TLS encryption and Route 53 manages domain routing. This architecture provides a secure, dependable, and responsive application environment that meets current demands and can grow with future demands.

## 5.5 Continuous Integration & Continuous Deployment

### 5.5.1 Overview

This project utilises GitHub Actions to create a Continuous Integration and Continuous Deployment (CI/CD) pipeline to maintain high code quality, security, and dependability while streamlining the development lifecycle. CI/CD automates testing, code validation, and deployment, reducing manual work and human error. Code changes sent to a version control system are automatically built and tested, preventing bugs from production and ensuring seamless integration with the current codebase. CD automates the release process, only deploying new code after successful checks. GitHub Actions interface with GitHub allows for automatic workflows, promoting an agile development process and preserving code integrity. CI/CD implementation also reduces time to deploy, increases system dependability, and allows zero downtime improvements through automatic CloudFront cache invalidation and backend process management with PM2.

### 5.5.2 Backend

#### *5.5.2.1 Security*

A specialised security pipeline was set up using GitHub Actions, combining SonarCloud for static code analysis and Snyk for dependency vulnerability scanning, to protect the backend codebase from known vulnerabilities and preserve a high degree of code quality. This pipeline makes sure security checks are a part of the continuous integration process by running automatically on all pull requests and on every push to the main or development branches.



A Snyk scan is a crucial task in the security workflow, comparing dependencies to a regularly updated vulnerability database. This is essential for Node.js apps, which often depend on third party packages that could introduce unnoticed security vulnerabilities. Snyk is installed globally and run using the `snyk test` command with the `severitythreshold=high` flag in the GitHub Actions

process. This strategy combines rigorous security enforcement with developer flexibility, allowing teams to gradually resolve noncritical concerns while recording medium and low risks (Snyk, 2023).

Performance on many builds is enhanced by using a caching method to put the Snyk CLI in the GitHub runners NPM cache. To authenticate every scan, the Snyk token is safely introduced into the environment using GitHub Secrets.

Because flaws in packages like authentication libraries, request handlers, or cryptography tools could compromise critical healthcare data and break industry compliance standards like HIPAA and GDPR, automated dependency scanning is essential for modern web applications (OWASP, 2021).

The screenshot displays the Snyk Code Analysis dashboard. At the top, the project path is 'Liam-Ronan-dev > Projects > Liam-Ronan-dev/Major-Project-Backend (main)'. The dashboard is divided into several sections: 'Overview', 'History', and 'Settings'. The 'Overview' section shows the project's status, including 'Created Sat 1st Feb 2025', 'Snapshot for commit 4193af9', and 'taken by snyk.io 18 hours ago'. It also displays the 'IMPORTED BY' (Liam Ronan), 'PROJECT OWNER' (Add a project owner), 'ENVIRONMENT' (Add a value), and 'BUSINESS CRITICALITY' (Add a value). The 'LIFECYCLE' section shows 'Add a value'. The 'ANALYSIS SUMMARY' indicates '40 analyzed files (74%) Repo breakdown'. The 'Issues' section shows '5 of 7 issues' with a search bar and filters for 'SEVERITY' (High, Medium, Low), 'PRIORITY SCORE' (0-1000), 'STATUS' (Open, Ignored), 'LANGUAGES' (JavaScript), and 'VULNERABILITY TYPES' (Improper Type Validation, Cleartext Transmission, Cross-Site Request Forgery). The main content area displays two 'Improper Type Validation' issues, both with a score of 472. The first issue is located in 'src/controllers/prescription.js' and describes a potential crash due to uncontrolled object properties. The second issue is also in 'src/controllers/prescription.js' and describes a potential crash due to uncontrolled object properties. Both issues include a '3 steps in 1 file' link to learn how to fix the issue.

Figure 41 Snyk Dashboard

## SonarCloud – Static Code Analysis

SonarCloud is a tool used for static code analysis and dependency scanning in backend projects. It examines code for vulnerabilities, code smells, security hotspots, and flaws. It also ensures code consistency and industry standards compliance. The SonarCloud scan uses Git history metadata and requires a valid SONAR\_TOKEN in GitHub Secrets. The dashboard provides insights, graphing, and issue tracking.

In conclusion, this integration helps to decrease technical debt and improve long term maintainability by encouraging the early detection of problems that traditional testing could miss, such as unused code, unhandled exceptions, and excessively complex routines (SonarSource, 2023).

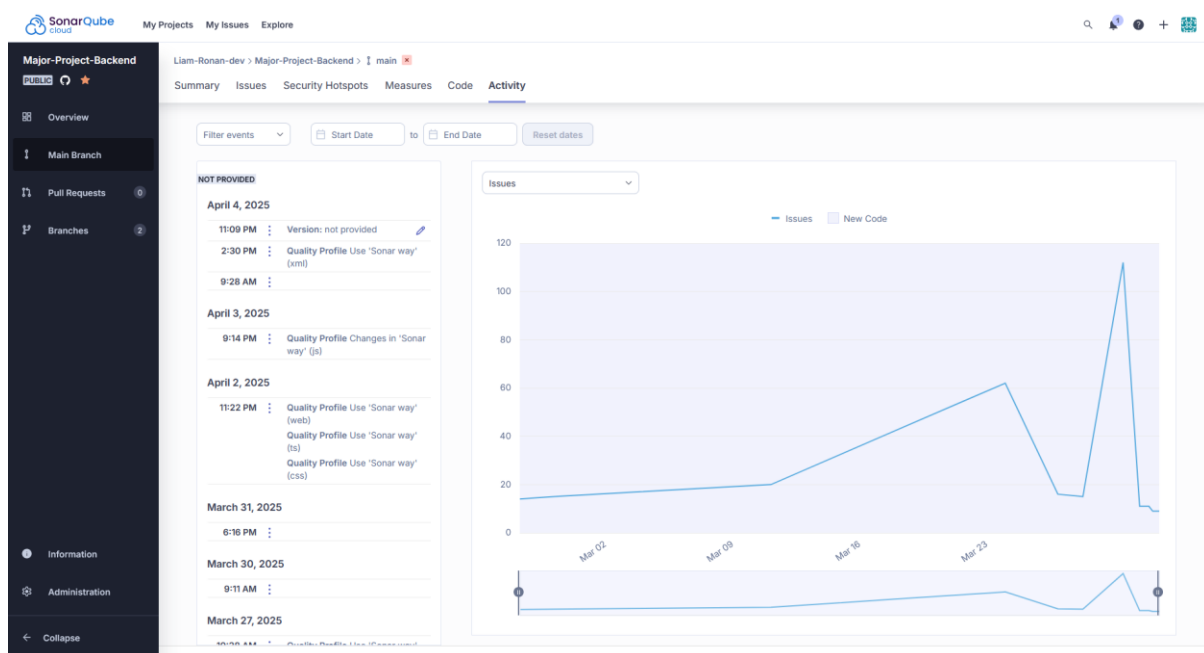


Figure 42 SonarCloud Dashboard

## Advantages of Automated Security Testing:

There are various advantages to integrating Snyk and SonarCloud into the CI pipeline:

- Shift left security lowers the cost of addressing problems by identifying them early in the development process.
- Productivity of developers: Without compromising code quality or security, automated scans free up developers to concentrate on creating features.
- Continuous assurance: The system automatically confirms that new code satisfies security and quality criteria with each commit and pull request.
- Increased visibility: Progress and project health audits are made simple by the centralisation of results in GitHub and SonarCloud dashboards.

### 5.5.2.2 Unit Testing & Code Quality

Using GitHub Actions, a code quality and unit testing pipeline was set up to ensure code reliability, preserve a consistent codebase, and enforce development standards. Every push and pull request to the main and development branches triggers this pipeline, which automates several jobs such as formatting validation, linting, standard commit enforcement, and unit testing.

By identifying problems early in the development cycle and making sure that only thoroughly tested and formatted code makes it to production, this method complies with Continuous Integration (CI) best practices.

```
name: 'Run Jest Tests & Format Code'

on:
  pull_request:
    branches:
      - main
      - develop
  push:
    branches:
      - main
      - develop

jobs:
  test-and-format:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [22.x] # Test on multiple Node.js versions

    steps:
      - name: 'Checkout Code'
        uses: actions/checkout@v4

      - name: 'Set up Node.js ${{ matrix.node-version }}'
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node-version }}
          cache: 'npm'

      - name: 'Install Dependencies'
        run: npm ci

      # Ensures commit messages follow conventions - Checks all commits, PRs, pushes
      - name: 'Validate commit messages with CommitLint'
        if: github.event_name == 'push'
        run: npx commitlint --last --verbose

      - name: 'Check Code Formatting with Prettier'
        run: npm run format

      - name: 'Check ESLint'
        run: npm run lint

      - name: 'Run Jest Tests'
        run: npm run test
```

Figure 43 CI Pipeline



## Code Quality Tools: ESLint, Prettier, and commitLint:

ESLint is a tool used in the pipeline to analyse code for errors, antipatterns, and stylistic issues, preventing runtime errors and ensuring consistent writing style. Prettier enhances ESLint by automatically formatting code according to a style guide, allowing developers to focus on logic rather than formatting details. CommitLint verifies commit messages according to the Conventional Commits specification, ensuring a clear Git history and making versioning and changelog generation more reliable.

## Unit Testing with Jest

Jest is a JavaScript testing framework that confirms code unit accuracy and ensures functionality without interference. It runs test suites in the backend, validating business logic, data manipulation, and API behaviour separately. Jest provides higher test coverage, simpler debugging, and increased developer confidence. It also ensures cross version compatibility and lowers risk when updating runtime environments.

## Benefits of Automated Code Quality and Unit Testing

Incorporating these quality checks and tests into the CI pipeline offers numerous benefits:

- Early error detection: By identifying problems before code is merged, regressions and logic errors are kept out of production.
- Enforced consistency: Linting and formatting rules ensure a uniform codebase, making the project easier to read and maintain.
- Better teamwork: Clear code and structured commit messages facilitate improved team and external contributor communication.

### 5.5.2.3 Deployment

A GitHub Actions deployment pipeline was developed to deploy directly to an Amazon EC2 instance running the Node.js API to automate the release of backend code to the production environment. As the last phase of the Continuous Deployment (CD) lifecycle, this procedure guarantees that new backend versions may be released effectively, reliably, and with the least amount of downtime.

Pushes to the main or develop branches initiate the deployment pipeline, which is run on an EC2 instances selfhosted GitHub Actions runner. This eliminates the need for SSH based

processes or third party CI runners by enabling direct access to local services like PM2 and NGINX.

```
name: Deploy Back-end to EC2 Instance

on:
  push:
    branches:
      - main
      - develop

jobs:
  deploy:
    runs-on: self-hosted

    strategy:
      matrix:
        node-version: [22.x]

    steps:
      - name: Checkout Code
        uses: actions/checkout@v4

      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node-version }}
          cache: 'npm'

      - name: Install modules
        run: npm ci

      # Append all secrets to .env file
      - name: 'Set Environment Variables'
        run: |
          cat <<EOF > .env
          PROD_ENV=${{ secrets.PROD_ENV }}
          ENCRYPTION_KEY=${{ secrets.ENCRYPTION_KEY }}
          JWT_REFRESH_TOKEN_SECRET=${{ secrets.JWT_REFRESH_TOKEN_SECRET }}
          JWT_REFRESH_TOKEN_SECRET_EXPIRES_IN=${{ secrets.JWT_REFRESH_TOKEN_SECRET_EXPIRES_IN }}
          JWT_SECRET=${{ secrets.JWT_SECRET }}
          JWT_SECRET_EXPIRES_IN=${{ secrets.JWT_SECRET_EXPIRES_IN }}
          MONGO_URI=${{ secrets.MONGO_URI }}
          PORT=${{ secrets.PORT }}
          TEMP_TOKEN_EXPIRES_IN=${{ secrets.TEMP_TOKEN_EXPIRES_IN }}
          ADMIN_EMAIL=${{ secrets.ADMIN_EMAIL }}
          ADMIN_EMAIL_PASSWORD=${{ secrets.ADMIN_EMAIL_PASSWORD }}
          BACKEND_URL=${{ secrets.BACKEND_URL }}
          EOF

      # Restart PM2 and update env variables
      - name: 'Restart PM2 Service'
        run: pm2 restart BackendAPI --update-env
```

Figure 44 Deployment Pipeline

## Breakdown:

### 1. Checkout Code:

The job begins by pulling the latest code from the GitHub repository using the actions/checkout action. This ensures the most recent commit from the main or develop branch is available for deployment.

### 2. Setup Node.js Environment:

Using actions/setupnode, Node.js version 22.x is installed on the selfhosted runner. This guarantees that the environment used during deployment matches the one used in development and testing.

### 3. Install Dependencies:

npm ci installs dependencies exactly as specified in the package-lock.json file. This method is faster and more reliable than npm install, especially in CI environments where clean, reproducible builds are essential.

### 4. Set Environment Variables:

Sensitive credentials and configuration values (e.g., JWT secrets, MongoDB URI, encryption keys) are injected securely into a .env file using GitHub Secrets. This ensures:

- Credentials never appear in source control.
- The deployed app has access to all runtime environment variables needed for secure operation.
- Configuration changes can be managed without modifying code.

Managing secrets through GitHub Actions is a secure and centralised method of configuration management

### 5. Restart PM2 Service:

Finally, the backend service is restarted using pm2 restart with the updateenv flag to apply the updated environment variables. PM2 is a Node.js process manager that ensures the API remains alive, logs errors, and supports zero downtime restarts (PM2 Docs, 2024). Restarting the service after deployment ensures that the latest code is deployed, and new environment configurations are loaded.

### Benefits of Automated Backend Deployment

- Consistency: Human error is minimised because every deployment follows the same set of procedures.
- Speed: After being merged, changes can be made public in a matter of seconds.
- Security: GitHub Secrets is used to safely manage secrets and environment variables.
- Traceability: Every deployment has a commit associated with it, allowing for complete visibility and rollback capabilities.

## 5.5.3 Frontend

### 5.5.3.2 End to end Testing

Using Playwright, an open-source testing framework created by Microsoft, an end-to-end (E2E) testing pipeline was put in place to make sure the Secure Prescription Systems frontend functions properly from the user's point of view. Every push to the main and development branches triggers this pipeline, which is integrated with GitHub Actions.

By starting real browser instances and verifying the UIs functional and visual behaviour, end to end testing replicates real user interactions with the application, including navigating pages, logging in, and submitting forms (Playwright, 2024).

```
name: Playwright Tests

on:
  push:
    branches:
      - main
      - develop

jobs:
  test:
    name: Run Playwright Tests
    timeout-minutes: 60
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Repository
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: 22
          cache: 'npm'

      - name: Install Dependencies
        run: npm ci

      - name: Cache Playwright Browsers
        uses: actions/cache@v4
        with:
          path: ~/.cache/ms-playwright
          key: playwright-${{ runner.os }}
          restore-keys: |
            playwright-

      - name: Install Playwright Browsers (If Not Cached)
        run: npx playwright install --with-deps

      - name: Run Playwright E2E Tests
        run: npm run e2e

      - name: Upload Playwright Test Report
        uses: actions/upload-artifact@v4
        if: ${{ !cancelled() }}
        with:
          name: playwright-report
          path: playwright-report/
          retention-days: 30
```

Figure 45 E2E Testing Pipeline

## Breakdown of the Playwright Test Workflow:

### 1. Checkout Repository

In order to make the entire project source code available for testing, the pipeline begins by cloning the code repository using `actions/checkout@v4`.

### 2. Node.js

`Actions/setupnode` is used to install Node.js version 22 on the runner environment. This provides compatibility with JavaScript/TypeScript test scripts and the frontend build

### 3. Set Up Dependencies

Based on the `package-lock.json` file, `npm ci` is used to install all frontend dependencies, guaranteeing a clean and repeatable dependency installation. For consistent test environments, this is crucial.

### 4. Playwright Browser Cache

The workflow uses the `actions/cache` action to cache Playwrights browser binaries (Chromium, Firefox, and WebKit) in order to enhance build efficiency. Test setup times are significantly reduced because Playwright browsers do not need to be downloaded again in the case of a cache hit.

### 5. Install browsers for Playwright

Playwright uses the `npx playwright install withdeps` command to install the required browser engines if the cache is out of date or non-existent. This guarantees that the most recent compatible browsers and system prerequisites are present in the test environment (Playwright, 2024).

### 6. Execute the E2E tests for Playwright

The Playwright test suite is executed by `npm run e2e` after the environment is prepared. These tests validate important user journeys like these by simulating actual interactions in a headless browser:

- Authentication
- Role based access control (RBAC)
- Viewing and managing prescriptions
- UI Accessibility and form handling

### 7. Upload playwright test report

Lastly, `actions/uploadartifact` is used to upload the test findings as a downloadable artifact. Visual traces, error logs, and test summaries are all included in the report. This artifact, which is kept for 30 days, aids developers in identifying errors and guarantees responsibility in quality assurance.

### **Benefits of Automated End to End Testing:**

The following benefits come from integrating Playwright E2E testing into a CI/CD pipeline:

- **Comprehensive Validation:** Tests assists identify UI problems that unit tests might overlook by simulating actual user interactions across several browsers (Microsoft, 2024).
- **Early Detection:** Defects are found as soon as code is pushed, which lowers the expense of repairing them later on in the development process.
- **Cross Browser Support:** Playwright ensures consistent behaviour across platforms by supporting WebKit, Chromium, and Firefox (Playwright, 2024).

#### *5.5.3.3 Unit Testing & Code Quality*

First, a Node.js environment with cached dependencies is set up and the most recent code is checked out. After that, it uses ESLint to find any possible problems or antipatterns and Prettier to check the code formatting. These procedures prevent frequent errors during development and maintain consistency.

This pipeline makes use of Vitest a testing framework made for frontend projects that are quick and easy to use, for unit testing. Vitest is perfect for projects built with React and Vite because it provides tight integration with Vite and executes tests in a quick, lightweight development environment (Vitest, 2024).

```
name: Continuous Integration

on:
  push:
    branches:
      - main
      - develop

jobs:
  lint-format-test:
    name: Lint, Format, and Test
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Repository
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: 22
          cache: 'npm'

      - name: Install Dependencies
        run: npm ci

      - name: Run Prettier Formatting Check
        run: npm run format

      - name: Run ESLint
        run: npm run lint

      - name: Run Vitest
        run: npm run test:unit
```

Figure 46 CI Pipeline

### 5.5.3.4 Deployment

To automate the release of the Secure Prescription Systems, React based frontend, a dedicated deployment pipeline was created using GitHub Actions. This pipeline builds and deploys the compiled frontend to an Amazon S3 bucket and, if pushed to the production branch, also triggers a CloudFront invalidation to refresh cached content. It ensures fast, secure, and reliable updates to both the development and production environments without any manual intervention.

```
name: Deploy Front-end to S3

on:
  push:
    branches:
      - develop # Deploy to Dev S3 bucket
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      # Checkout code
      - name: Checkout Code
        uses: actions/checkout@v3

      # Configure Credentials
      - name: Configure AWS Credentials
        uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${ secrets.AWS_ACCESS_KEY_ID }
          aws-secret-access-key: ${ secrets.AWS_SECRET_ACCESS_KEY }
          aws-region: eu-west-1

      # Install modules - Clean Installation
      - name: Install modules
        run: npm ci

      # Build front-end
      - name: Build Front-end
        run: npm run build

      # Determine env
      - name: Determine environment
        run: |
          if [[ "${ github.ref }" == "refs/heads/main" ]]; then
            echo "BUCKET_ID=health-service.click" >> $GITHUB_ENV
            echo "RUN_INVALIDATION=true" >> $GITHUB_ENV
            echo "DISTRIBUTION_ID=${ secrets.DISTRIBUTION_ID }" >> $GITHUB_ENV
          else
            echo "BUCKET_ID=dev-health-service-app" >> $GITHUB_ENV
            echo "RUN_INVALIDATION=false" >> $GITHUB_ENV
          fi

      # Deploy to S3 Bucket
      - name: Deploy to S3
        run: aws s3 sync ./dist/ s3://$BUCKET_ID --delete

      # Only runs if the deploy is to main # Prod
      - name: Create CloudFront invalidation
        if: env.RUN_INVALIDATION == 'true'
        run: aws cloudfront create-invalidation --distribution-id ${ secrets.DISTRIBUTION_ID }
    } --paths "/*"
```

Figure 47 Deployment Pipeline



## Breakdown of the Frontend Deployment Workflow

### 1. Checkout code

The pipeline starts by using the actions/checkout action to pull the most recent version of the code from the repository, providing the runner with access to the frontends complete source code.

### 2. Configure Amason credentials

The pipeline makes use of the awsactions/configureawscredentials action to communicate securely with AWS services. In order to guarantee that credentials are never hardcoded in the workflow and are encrypted both in transit and at rest, it authenticates using access keys kept in GitHub Secrets.

### 3. Install dependencies

Using npm ci, the projects dependencies are installed cleanly, guaranteeing that the same versions listed in package-lock.json are used. This lowers the possibility of environmentspecific problems and facilitates deterministic builds.

### 4. Build Frontend

Using Vite, the application is built by running npm run build, which creates static HTML, CSS, and JavaScript files in the dist directory from the React codebase. These are prepared for online deployment and have been performanceoptimised (Vite, 2024).

### 5. Determine Environment

Whether the push was made to the main branch, or the development branch is verified using a specially written conditional script. Depending on the outcome:

- If the branch is main, it signifies the pipeline to cause a CloudFront invalidation and sets the environment variables to deploy to the production S3 bucket (healthservice.click).
- If the branch is develop, it deploys to the development bucket (devhealthserviceapp) without triggering a CDN refresh. This supports proper staging workflows and minimises unnecessary cache busting.

### 6. Deploy to Amason S3

The AWS CLI command `aws s3 sync ./dist/ s3://$BUCKET_ID delete` is used to upload the compiled frontend assets to the relevant S3 bucket. By replacing or deleting files in the destination bucket to match the local dist/ subdirectory, this operation completes a full synchronisation. For hosting static webpages, Amason S3 provides a dependable, scalable, and highly accessible object storage solution.

### 7. CloudFront Invalidation (Production Only)

AWS CloudFront createInvalidation is used to send an invalidation request to the CloudFront distribution if the push was to the main branch. In order to force the most recent frontend build to be served to visitors worldwide, this makes sure that all edge locations clear their cached content (/ \* path).

**Benefits of Automated Frontend Deployment:**

- **Efficiency:** Changes are distributed quickly following a push, removing the need for manual uploads or intervention.
- **Consistency:** Because of automation, the build and deployment procedures are repeatable and errorfree.
- **Scalability and Speed:** When combined, S3 and CloudFront provide static assets with low latency and high availability on a global scale.

## 5.6 Development

### 5.6.1 Backend

#### *5.6.1.1 Project Structure and Initial Setup*

Using a modular architecture to promote scalability, separation of concerns, and maintainability, the Secure Prescription Systems backend was developed with Node.js and Express. The applications code is arranged into important directories under src/, along with several configurations at the root level, as seen in the project structure figure 48.

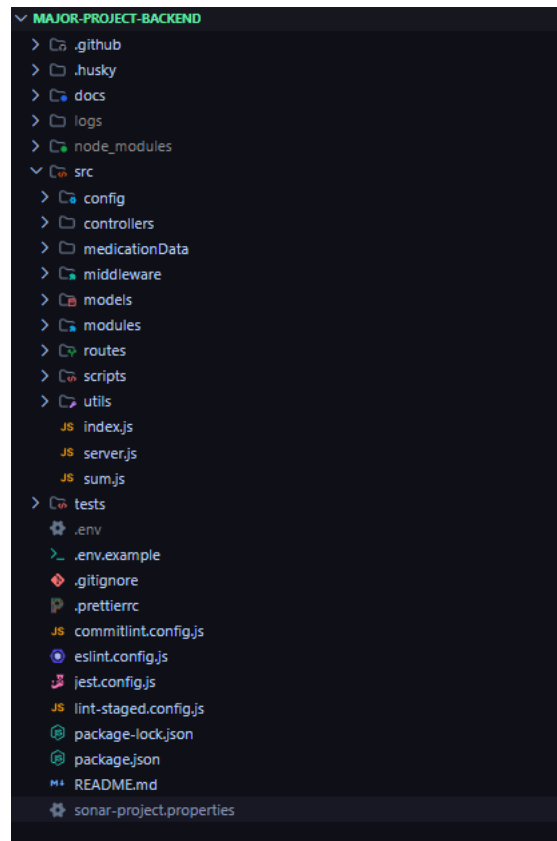


Figure 48 Backend Folder Structure

## Folder Structure and Modular Design

- .github/: contains the testing, security and deployment pipelines that were outlined above.
- .husky/: contains the git hooks such as the precommit hook for running lintstaged which in turn will run the linters, formatter etc. Also, the prepush hook resonates here with the ability to run the tests before pushing the code. If any of these hooks fail the checks, they will stop the code from being pushed.
- The core of the backend logic is the src/ directory, which has subfolders with various functions:
- Controllers/: Includes all of the business logic needed to respond to incoming API requests. This covers tasks including updating patients records, registering users, and filling prescriptions.
- Routes/: Ensures a clear separation between functionality and routing by mapping HTTP endpoints to controller methods while adhering to RESTful API standards.
- Models/: Mongoose schema definitions are stored under models/ . Core entities including User, Patient, Prescription, Medication, and Appointment are included

here. The format and constraints on the data stored in MongoDB are specified by each schema.

- **Middleware/:** Logging, error handling, authorisation, and security related tasks like role permissions and JWT validation are all handled via reusable middleware.
- **Config/:** Reusable configuration logic, including CORS settings and allowed origins, database connection logic (db.js), and email sending features, are contained here.
- **Utils/:** Contains utility functions like encryption for patients and prescriptions, validators to ensure the data follows the correct format before sending requests, and a prescription seeder to populate the prescriptions model with realistic data.
- **Scripts/:** These are used for onetime actions like seeding medicinal products data from external, approved sources into the database.
- **medicationData/:** Contains the original xml file with all the medicines that have been assessed by the Health Products Regulatory Authority (HPRA) (Data.gov.ie, 2018).
- **Modules/:** Hold the reusable authentication functions for hashing passwords and license numbers, creating JWTs, and ensuring users are authenticated to request a certain resource.

By organising functions and allowing independent testing or updates for specific components, this structure enhances development workflow.

## Database Model example: Prescription

The most important models are the prescription and item model which have a one to many relationship as in a prescription may have many items associated with it. It defines relationships to other core entities such as Doctor, Pharmacist, and Patient. Notably, it includes custom fields for prescription status, automatically generated IDs, and encryption.

```

import mongoose from 'mongoose';
import { encryptData, decryptData } from '../utils/encryption.js';

const PrescriptionSchema = new mongoose.Schema({
  prescriptionId: {
    type: String,
    unique: true,
  },
  doctorId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true,
  },
  pharmacistId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true,
  },
  patientId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Patient',
    required: true,
  },
  items: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Item',
    },
  ],
  status: {
    type: String,
    enum: ['Assigned', 'Pending', 'Processed', 'Completed', 'Cancelled'],
    default: 'Assigned',
  },
  notes: {
    type: String,
    set: encryptData,
    get: decryptData,
  },
  createdAt: {
    type: Date,
    default: Date.now,
  },
  updatedAt: {
    type: Date,
    default: Date.now,
  },
});

// Auto-generate `prescriptionId` if missing
PrescriptionSchema.pre('save', function (next) {
  if (!this.prescriptionId) {
    this.prescriptionId = new mongoose.Types.ObjectId().toHexString();
  }
  next();
});

// Enable automatic decryption when retrieving data
PrescriptionSchema.set('toJSON', { getters: true });
PrescriptionSchema.set('toObject', { getters: true });

export const Prescription = mongoose.model('Prescription', PrescriptionSchema);

```

Figure 49 Prescription Model

```

import mongoose from 'mongoose';
import { encryptData, decryptData } from '../utils/encryption.js';

const ItemSchema = new mongoose.Schema({
  prescriptionId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Prescription',
    required: true,
  },
  medicationId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Medication',
    required: true,
  },
  specificInstructions: {
    type: String,
    required: true,
    set: encryptData,
    get: decryptData,
  },
  repeats: {
    type: Number,
    default: 0,
  },
  dosage: {
    type: String,
    required: true,
    set: encryptData,
    get: decryptData,
  },
  amount: {
    type: String,
    required: true,
    set: encryptData,
    get: decryptData,
  },
  pharmacistNote: {
    type: String,
    default: null,
    set: encryptData,
    get: decryptData,
  },
});

ItemSchema.set('toJSON', { getters: true });
ItemSchema.set('toObject', { getters: true });

export const Item = mongoose.model('Item', ItemSchema);

```

Figure 50 Item Model

Any notes and items that are added to a prescription are encrypted at the time of saving and immediately decrypted upon retrieval using bespoke getter and setter functions. By utilising AES256 encryption integrated into a utility function through Nodes crypto module, this provides compliance to data protection regulations by rendering sensitive data in the database unidentifiable.

## Server Configuration and Entry Point

The Express server is started on an assigned port by server.js, which loads environment variables using dotenv. Index.js manages the primary server logic, including middleware setup, route configuration, realtime notifications initialisation, and running the connectDb function.

```
import * as dotenv from 'dotenv';
import mongoose from 'mongoose';

dotenv.config();

export const connectDB = async () => {
  const dbURI = process.env.MONGO_URI;

  try {
    await mongoose.connect(dbURI);
    console.log(`MongoDB connected successfully!`);
  } catch (error) {
    console.error(`Failed to connect to Database: ${error}`);
  }
};

export const disconnectDB = async () => {
  try {
    await mongoose.connection.close();
    console.log('MongoDB disconnected successfully');
  } catch (error) {
    console.error(`Error disconnecting from MongoDB: ${error}`);
  }
};
```

Figure 51 Database Connection Function

```

const app = express();
export const server = http.createServer(app);

// Initialise socket.io
export const io = new Server(server, {
  cors: {
    origin: allowedOrigins,
    credentials: true,
    allowedHeaders: ['Content-Type', 'Authorization', 'Origin', 'Accept'],
  },
});

// Store connected users: Map of userId => socket.id
export const connectedUsers = new Map();

// Socket.io connection
io.on('connection', (socket) => {
  console.log(`Client connected: ${socket.id}`);

  socket.on('register', (userId) => {
    connectedUsers.set(userId, socket.id);
    console.log(`Registered user ${userId}`);
  });

  socket.on('disconnect', () => {
    for (const [userId, id] of connectedUsers.entries()) {
      if (id === socket.id) {
        connectedUsers.delete(userId);
        console.log(`User ${userId} disconnected`);
        break;
      }
    }
  });
});

// Use a logger to display request status code, origin, time/date etc
app.use(logger);

// Allowing the Front-end to make requests to the Backend API
app.use(cors(corsOptions));

// Apply Helmet to Secure HTTP Headers
app.use(helmet());

app.use(express.json()); // Parse JSON Requests
app.use(cookieParser()); // Parse cookies
app.use(express.urlencoded({ extended: true }));

app.get('/api/health', (req, res) => {
  res.json({ message: 'hello world!' });
});

// Use routes
app.use('/api', userRoutes);
app.use('/api/admin', adminRoutes);
app.use('/api', prescriptionRoutes);
app.use('/api', appointmentRoutes);
app.use('/api', patientRoutes);
app.use('/api', medicationRoutes);

// Forward errors to the error logger
app.use(errorHandlerLogger);

// Using the reusable database connection function
connectDB();

export default app;

```

Figure 52 Application Starting Point



```
import * as dotenv from 'dotenv';
import { server } from './index.js';

dotenv.config();

const PORT = process.env.PORT || 3000;

// Run the Server
server.listen(PORT, () => {
  console.log(`Server listening on http://localhost:${PORT}`);
});
```

Figure 53 Server.js

Additionally, it registers all API routes including endpoints for patients, users, prescriptions, under a single /api namespace. A connectedUsers map is used to monitor active sessions, and Socket.io is initially configured to facilitate realtime notifications between doctors and pharmacists.

Lastly, the credentials specified in the .env file are used by the connectDB() function to create a connection to the MongoDB database. To confirm server availability during deployment or continuous integration checks, a basic /api/health endpoint is also offered.

For secure and effective request handling, realtime communication, and database integration, this structure offers a clear and modular structure.

## Environment Configuration and Secrets Management

```
PORT=3001
MONGO_URI=
JWT_SECRET=""
JWT_SECRET_EXPIRES_IN=""
ENCRYPTION_KEY=""
```

Figure 54 env Example

- MONGO\_URI – MongoDB Atlas connection string
- ENCRYPTION\_KEY – AES key used for field level encryption
- JWT\_SECRET – Used for signing access tokens
- ADMIN\_EMAIL and ADMIN\_EMAIL\_PASSWORD – For sending notifications via Nodemailer

### 5.6.1.2 Database Models and Relationships

The Secure Prescription Systems backend makes use of Mongoose, an Object Data Modelling (ODM) module for MongoDB, to manage data persistence. Mongoose is perfect for keeping an organised prescription database system since it enables the establishment of tightly typed schemas with validation rules, default values, timestamps, and relationships between documents.

#### User Model:

```
import mongoose from 'mongoose';

const ROLES = {
  DOCTOR: 'doctor',
  PHARMACIST: 'pharmacist',
};

const UserSchema = new mongoose.Schema({
  email: {
    type: String,
    required: true,
    unique: true,
  },
  password: {
    type: String,
    required: true,
  },
  role: {
    type: String,
    enum: [ROLES.DOCTOR, ROLES.PHARMACIST],
    required: true,
  },
  licenseNumber: {
    type: String, // Unique license/registration #
    required: true,
    unique: true,
  },
  isVerified: {
    type: Boolean,
    default: false, // will be set to true after admin verification
  },
  verificationToken: {
    type: String, // Stores the one-time token sent to the admin
  },
  verificationTokenExpires: {
    type: Date,
  },
  mfaEnabled: {
    type: Boolean,
    default: true,
  },
  mfaSecret: {
    type: String,
  },
});

export const User = mongoose.model('User', UserSchema);
```

Figure 55 User Model

Specifically, "Doctor" and "Pharmacist" are supported by the User schema by an enum field. Also, it supports multifactor authentication (MFA) with parameters like mfaEnabled and mfaSecret. Email verification and secure account setup are handled by additional parameters like isVerified, verificationToken, and verificationTokenExpires. Since both doctors and pharmacists stem from this core structure, the schema is essential to the system.

## Patient Model:

```
import mongoose from 'mongoose';
import { encryptData, decryptData } from '../utils/encryption.js';

const PatientSchema = new mongoose.Schema({
  firstName: {
    type: String,
    required: true,
    set: encryptData,
    get: decryptData,
  },
  lastName: {
    type: String,
    required: true,
    set: encryptData,
    get: decryptData,
  },
  dateOfBirth: {
    type: String,
    required: true,
    set: encryptData,
    get: decryptData,
  },
  gender: {
    type: String,
    required: true,
    set: encryptData,
    get: decryptData,
  },
  phoneNumber: {
    type: String,
    required: true,
    set: encryptData,
    get: decryptData,
  },
  email: {
    type: String,
    unique: true,
    sparse: true,
    set: encryptData,
    get: decryptData,
  },
  address: {
    street: { type: String, set: encryptData, get: decryptData },
    city: { type: String, set: encryptData, get: decryptData },
    postalCode: { type: String, set: encryptData, get: decryptData },
    country: {
      type: String,
      required: true,
      set: encryptData,
      get: decryptData,
    },
  },
  medicalHistory: [
    {
      condition: { type: String, set: encryptData, get: decryptData },
      diagnosedAt: { type: Date },
      notes: { type: String, set: encryptData, get: decryptData },
    },
  ],
  doctorId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true,
  },
  prescriptions: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Prescription' }],
  appointments: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Appointment' }],
  emergencyContact: {
    name: { type: String, set: encryptData, get: decryptData },
    relationship: { type: String, set: encryptData, get: decryptData },
    phoneNumber: { type: String, set: encryptData, get: decryptData },
  },
  createdAt: { type: Date, default: Date.now },
});

PatientSchema.set('toJSON', { getters: true });
PatientSchema.set('toObject', { getters: true });

export const Patient = mongoose.model('Patient', PatientSchema);
```

Figure 56 Patient Model

Personal information including name, gender, date of birth, medical history, and emergency contact are all included in the patient schema. A doctorId references each patient to a doctor, creating a onetomany relationship (one doctor → many patients). This guarantees that doctors can only see and treat the patients they have been allocated.

### Prescription Model:

```
import mongoose from 'mongoose';
import { encryptData, decryptData } from '../utils/encryption.js';

const PrescriptionSchema = new mongoose.Schema({
  prescriptionId: {
    type: String,
    unique: true,
  },
  doctorId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true,
  },
  pharmacistId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true,
  },
  patientId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Patient',
    required: true,
  },
  items: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Item',
    },
  ],
  status: {
    type: String,
    enum: ['Assigned', 'Pending', 'Processed', 'Completed', 'Cancelled'],
    default: 'Assigned',
  },
  notes: {
    type: String,
    set: encryptData,
    get: decryptData,
  },
  createdAt: {
    type: Date,
    default: Date.now,
  },
  updatedAt: {
    type: Date,
    default: Date.now,
  },
});

// Auto-generate `prescriptionId` if missing
PrescriptionSchema.pre('save', function (next) {
  if (!this.prescriptionId) {
    this.prescriptionId = new mongoose.Types.ObjectId().toHexString();
  }
  next();
});

// Enable automatic decryption when retrieving data
PrescriptionSchema.set('toJSON', { getters: true });
PrescriptionSchema.set('toObject', { getters: true });

export const Prescription = mongoose.model('Prescription', PrescriptionSchema);
```

Figure 57 Prescription Model

ObjectId references (doctorId, pharmacistId, and patientId) link prescriptions to three parties: the patient, the doctor, and the pharmacist. Prescriptions have embedded item references for medications prescribed as well as a status field with an enum structure (e.g., Assigned, Pending, Processed). AES256 encryption is used in this models notes field to secure sensitive data.

Using the Mongoose middleware (pre(save)), a prescriptionId is automatically generated as a distinct string identification that is distinct from MongoDBs native \_id.

### Medication Model:

```
import mongoose from 'mongoose';

const MedicationSchema = new mongoose.Schema({
  name: { type: String, required: true },
  activeSubstance: { type: String },
  authorisationNumber: { type: String },
  atcCode: { type: String },
  routeOfAdministration: { type: String },
  productId: { type: String, unique: true },
  createdAt: { type: Date, default: Date.now },
});

export const Medication = mongoose.model('Medication', MedicationSchema);
```

*Figure 58 Medication Model*

Name, activeSubstance, authorisationNumber, and routeOfAdministration are among the structured medical data fields in the Medication schema that are extracted from HPRA verified API. When creating prescriptions, this dataset which is read only within the system is utilised for searching medications.

## Item Model:

```
import mongoose from 'mongoose';
import { encryptData, decryptData } from '../utils/encryption.js';

const ItemSchema = new mongoose.Schema({
  prescriptionId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Prescription',
    required: true,
  },
  medicationId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Medication',
    required: true,
  },
  specificInstructions: {
    type: String,
    required: true,
    set: encryptData,
    get: decryptData,
  },
  repeats: {
    type: Number,
    default: 0,
  },
  dosage: {
    type: String,
    required: true,
    set: encryptData,
    get: decryptData,
  },
  amount: {
    type: String,
    required: true,
    set: encryptData,
    get: decryptData,
  },
  pharmacistNote: {
    type: String,
    default: null,
    set: encryptData,
    get: decryptData,
  },
});

ItemSchema.set('toJSON', { getters: true });
ItemSchema.set('toObject', { getters: true });

export const Item = mongoose.model('Item', ItemSchema);
```

Figure 59 Item Model

Multiple items (medication entries) with fields for dosage, amount, repeats, and pharmacistNote may be included in a single prescription. Every item is associated with a specific prescription (prescriptionId) and refers to a medication. A single prescription can have multiple independently trackable items due to this structure.

## Appointment Model:

```
import mongoose from 'mongoose';

const AppointmentSchema = new mongoose.Schema({
  doctorId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true,
  }, // The doctor assigned
  patientId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Patient',
    required: true,
  }, // The patient
  date: { type: Date, required: true }, // Date & time of the appointment
  status: {
    type: String,
    enum: ['Scheduled', 'Completed', 'Cancelled', 'No Show'],
    default: 'Scheduled',
    required: true,
  }, // Appointment status
  notes: { type: String, required: true }, // Notes from the doctor
  createdAt: { type: Date, default: Date.now }, // Timestamp for creation
  updatedAt: { type: Date, default: Date.now }, // Timestamp for updates
});

export const Appointment = mongoose.model('Appointment', AppointmentSchema);
```

Figure 60 Appointment Model

Doctors schedule appointments. Along with a date, status, and optional notes field, the model refers to both parties (doctorId, patientId). This makes it possible to keep track of user follow ups, consultations, and scheduled appointments.

### 5.6.1.3 Controllers and Route Definitions

A separate controller handles prescriptions, ensuring secure interactions between doctors and pharmacists over role based HTTP endpoints. Layered middleware and validators protect routes, enabling access control and strong request validation. Prescription.js controller file defines CRUD actions and pharmacist specific modifications, defining a document with related medication items and metadata.

## Creating Prescriptions:

CRUD actions and pharmacist specific modifications for prescriptions are defined in the prescription.js controller file. A prescription is essentially a document made up of related medication items and metadata (doctor, patient, and pharmacist).

Creating Prescriptions: Only authorised doctors are able to fill prescriptions. Using doctorId, the controller first confirms that the patient is the patient of the requesting doctor. After that, it generates the prescription and related item data, each of which refers to a distinct medication. To protect sensitive medical data while it is at rest, the prescription is encrypted using a getter setter pair that is specified in the schema.

```
try {
  const { patientId, pharmacistId, notes, items } = req.body;

  if (!patientId || !pharmacistId || !items?.length) {
    return res.status(400).json({ message: 'Missing required fields' });
  }

  // Ensure patient belongs to the doctor
  const patient = await Patient.findOne({ _id: patientId, doctorId: req.user.id });
  if (!patient) {
    return res.status(403).json({ message: 'You can only prescribe for your own patients' });
  }

  const prescription = await Prescription.create({
    doctorId: req.user.id,
    patientId,
    pharmacistId,
    notes,
  });

  // Create items
  const itemIds = await Promise.all(
    items.map(async (item) => {
      const { medicationId, dosage, amount, specificInstructions, repeats, pharmacistNote } =
        item;

      const medication = await Medication.findById(medicationId);
      if (!medication) throw new Error(`Medication not found: ${medicationId}`);

      const newItem = await Item.create({
        prescriptionId: prescription._id,
        medicationId,
        dosage,
        amount,
        specificInstructions,
        pharmacistNote,
        repeats: repeats ?? 0,
      });

      return newItem._id;
    })
  );

  prescription.items = itemIds;
  await prescription.save();
}
```

Figure 61 Create Prescription Function



## Item Creation Logic:

Every prescription has several items that correspond to different drugs. `Promise.all()` is used to insert them into the database, allowing asynchronous reference generation and collection. These item IDs are associated with the Prescription document after they are created.

## Retrieval & Population:

The `populate()` method is widely used to incorporate relevant user and medication data in the same query when retrieving prescriptions (`getAllPrescriptions` and `getPrescriptionById`). This removes the need for extra fetches, improving frontend rendering. To enable rich data in user interface displays, for instance, `prescription.items.medicationId` is filled with the names of medications and their active ingredients.

```
// Get All Prescriptions (Doctors & Pharmacists)
export const getAllPrescriptions = async (req, res) => {
  try {
    const role = req.user.role;
    const userId = req.user.id;

    const filter = role === 'doctor' ? { doctorId: userId } : { pharmacistId: userId };

    const prescriptions = await Prescription.find(filter)
      .populate('patientId', 'firstName lastName dateOfBirth')
      .populate('pharmacistId', 'email')
      .populate('doctorId', 'email')
      .populate({
        path: 'items',
        select: 'medicationId dosage amount specificInstructions pharmacistNote repeats'
      },
      {
        populate: {
          path: 'medicationId',
          select: 'name activeSubstance',
        }
      })
      .sort({ createdAt: -1 });

    res.status(200).json({ count: prescriptions.length, data: prescriptions });
  } catch (err) {
    console.error('Error fetching prescriptions:', err);
    res.status(500).json({ message: 'Failed to fetch prescriptions' });
  }
};
```

Figure 62 Get Prescriptions

## Pharmacists Status Updates:

Pharmacists Status Updates: Pharmacists can update the status and optional remarks on individual items via the PATCH mechanism. The controller verifies their position and makes sure that only allowed statuses (such as Processed and Completed) can be set. It modifies the prescription and related items appropriately if it is valid. Additionally, the schemas getter setter is used to safely store and retrieve encrypted data, such as notes.

```
const { id } = req.params;
const { status, notes, itemNotes } = req.body;

if (req.user.role !== 'pharmacist') {
  return res.status(403).json({ message: 'Only pharmacists can update prescriptions'
});
}

const allowedStatuses = ['Pending', 'Processed', 'Completed', 'Cancelled'];
if (status && !allowedStatuses.includes(status)) {
  return res.status(400).json({ message: 'Invalid status update' });
}

const prescription = await Prescription.findById(id);

if (!prescription) {
  return res.status(404).json({ message: 'Prescription not found' });
}

// Update status and notes if provided
if (status) prescription.status = status;
if (notes) prescription.notes = notes;

await prescription.save();
```

Figure 63 Pharmacist Update Function

```
// Update pharmacist notes on individual items
if (Array.isArray(itemNotes)) {
  await Promise.all(
    itemNotes.map(async ({ itemId, pharmacistNote }) => {
      const item = await Item.findOne({
        _id: itemId,
        prescriptionId: prescription._id,
      });

      if (item && pharmacistNote !== undefined) {
        item.pharmacistNote = pharmacistNote;
        await item.save();
      }
    })
  );
}
```

Figure 64 Pharmacist Update Item Note

## Prescription Route Definitions:

The middleware layers and endpoint structure for every prescription operation are specified in the routes/prescription.js file.

```
const router = express.Router();

// Doctor: Create prescription
router.post(
  '/prescriptions',
  ensureAuthenticated,
  authorizeRoles('doctor'),
  validateCreatePrescription,
  handleInputErrors,
  createPrescription
);

// Doctor: Update their own prescription
router.put(
  '/prescription/:id',
  ensureAuthenticated,
  authorizeRoles('doctor'),
  verifyOwnership('Prescription'),
  validateUpdatePrescription,
  handleInputErrors,
  updatePrescription
);

// Doctor: Delete their own prescription
router.delete(
  '/prescription/:id',
  ensureAuthenticated,
  authorizeRoles('doctor'),
  verifyOwnership('Prescription'),
  validateDeletePrescription,
  handleInputErrors,
  deletePrescription
);

// Doctor/Pharmacist: View all accessible prescriptions
router.get(
  '/prescriptions',
  ensureAuthenticated,
  authorizeRoles('doctor', 'pharmacist'),
  getAllPrescriptions
);

// Doctor/Pharmacist: View a single prescription
router.get(
  '/prescription/:id',
  ensureAuthenticated,
  authorizeRoles('doctor', 'pharmacist'),
  verifyOwnership('Prescription'),
  validateGetPrescriptionById,
  handleInputErrors,
  getPrescriptionById
);

// Pharmacist: Update prescription status and notes
router.patch(
  '/prescription/:id/status',
  ensureAuthenticated,
  authorizeRoles('pharmacist'),
  verifyOwnership('Prescription'),
  validatePatchPrescription,
  handleInputErrors,
  updatePrescriptionStatusAndNotes
);

export default router;
```

Figure 65 Prescription Route

Authenticated middleware guarantees that these routes are only accessible by logged in users.

```
// Protect Middleware
export const ensureAuthenticated = (req, res, next) => {
  const token = req.cookies.accessToken;

  if (!token) {
    return res.status(401).json({ message: 'Not authorized. Missing token' });
  }

  const decodedToken = verifyToken(token);

  if (!decodedToken) {
    return res.status(401).json({ message: 'Invalid or expired token' });
  }

  req.user = decodedToken;
  next();
};
```

Figure 66 Ensure Authenticated Middleware

RBAC is enforced by `authoriseRoles(...)`, which limits access according to user roles (e.g., pharmacist or doctor).

```
// Role-based access control middleware
export const authorizeRoles = (...roles) => {
  return (req, res, next) => {
    const userRole = req.user.role.toLowerCase();
    if (!roles.includes(userRole)) {
      return res.status(403).json({ message: 'Unauthorized access' });
    }
    next();
  };
};
```

Figure 67 Access Control Middleware

`verifyOwnership(Prescription)` verifies that the person logged in is, in fact, associated with the resource being accessed (e.g., the pharmacist assigned to the prescription or the doctor who owns it). By doing this, horizontal privilege escalation is avoided.

```

export const verifyOwnership = (modelType) => {
  return async (req, res, next) => {
    try {
      const { id } = req.params;

      const models = [
        Prescription,
        Patient,
        Appointment,
        Medication,
        Item,
      ];

      const model = models[modelType];
      if (!model) {
        return res.status(400).json({ message: 'Invalid model type' });
      }

      const record = await model.findById(id);
      if (!record) {
        return res.status(404).json({ message: `${modelType} not found` });
      }

      // Doctor can access prescriptions, patients, appointments they created
      if (req.user.role === 'doctor') {
        if (
          ['Prescription', 'Patient', 'Appointment'].includes(modelType) &&
          record.doctorId.toString() !== req.user.id
        ) {
          return res
            .status(403)
            .json({ message: 'Unauthorized: You do not own this ${modelType}' });
        }
      }

      // Pharmacist can access prescriptions/items assigned to them
      if (req.user.role === 'pharmacist') {
        if (modelType === 'Prescription' && record.pharmacistId.toString() !== req.user.id) {
          return res
            .status(403)
            .json({ message: 'Unauthorized: You are not assigned to this prescription' });
        }
      }

      if (modelType === 'Item') {
        // Load related prescription to verify pharmacist assignment
        const prescription = await Prescription.findById(record.prescriptionId);
        if (!prescription || prescription.pharmacistId.toString() !== req.user.id) {
          return res
            .status(403)
            .json({ message: 'Unauthorized: You are not assigned to this prescription item' });
        }
      }
    } catch (error) {
      console.error('Ownership verification error:', error);
      res.status(500).json({ message: 'Ownership verification failed' });
    }
  };
};

```

Figure 68 Verify Ownership Middleware

Incoming data is guaranteed to follow expected formats via validation middlewares such as `validateCreatePrescription` and `validatePatchPrescription`. These enforce criteria like required enum, and correct object ID references and check nested arrays (like items) using tools like `expressvalidator` and custom logic.

```
// Reusable validation for items in a prescription
const itemValidations = [
  body('items').isArray({ min: 1 }).withMessage(
    'Prescription must have at least one item.'),

  body('items.*.medicationId')
    .notEmpty()
    .withMessage('Each item must have a medication ID.')
    .bail()
    .isMongoId()
    .withMessage('Invalid medication ID.'),

  body('items.*.specificInstructions')
    .notEmpty()
    .withMessage('Specific instructions required.')
    .bail()
    .isString()
    .withMessage('Instructions must be a string.'),

  body('items.*.dosage')
    .notEmpty()
    .withMessage('Dosage is required.')
    .bail()
    .isString()
    .withMessage('Dosage must be a string.'),

  body('items.*.amount')
    .notEmpty()
    .withMessage('Amount is required.')
    .bail()
    .isString()
    .withMessage('Amount must be a string.'),

  body('items.*.repeats')
    .optional()
    .isInt({ min: 0 })
    .withMessage('Repeats must be a non-negative integer.'),

  body('items.*.pharmacistNote')
    .optional()
    .isString()
    .withMessage('Pharmacist note must be a string if provided.'),
];
```

Figure 69 Validation Middleware

Before reaching the controller code, `handleInputErrors` gathers validation errors and provides the correct error responses.

```
export const handleInputErrors = (req, res, next) => {
  const errors = validationResult(req);

  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  next();
};
```

Figure 70 Error Middleware

The routing structure ensures clean RESTful design. HTTP methods align with operations:

- POST /prescriptions – Create
- PUT /prescription/:id – Update
- DELETE /prescription/:id – Delete
- GET /prescriptions – List all (filtered by role)
- GET /prescription/:id – View one
- PATCH /prescription/:id/status – Update status and item notes (pharmacist only)

This design ensures granular access control, strict input validation, encrypted data handling, and secure role-based data flow throughout the lifecycle of a prescription.

#### 5.6.1.4 Authentication and Authorisation

The system uses industry standard procedures to guarantee that users can only carry out tasks relevant to their responsibilities and that only verified users have access to protected endpoints.

### Passwords and Sessions for Authentication:

Session based login with JWTs and hashed passwords are used to securely handle user credentials

```
// Create JWT - using JWT secret
export const createJWT = (user) => {
  const token = jwt.sign({ id: user._id, role: user.role }, process.env.JWT_SECRET, {
    subject: 'AccessAPI',
    expiresIn: process.env.JWT_SECRET_EXPIRES_IN,
  });
  return token;
};
```

Figure 71 Create JWT Function

Password Hashing: Bcrypt with salting is used to hash user passwords and license numbers during registration (hashField). This reduces the possibility of data breaches and stops unencrypted storage.

```
// Function for hashing fields into the database
export const hashField = async (field) => {
  const salt = await bcrypt.genSalt(saltRounds);
  return bcrypt.hash(field, salt);
};

// Function for comparing user credentials with hashed value in the database
export const compareField = async (field, hashedField) => {
  return await bcrypt.compare(field, hashedField);
};
```

Figure 72 Hashing Functions

Login Procedure: The users email address and password are verified during the login process. If successful, the users ID is saved in a secure, HTTP only cookie (mfa\_session) to start a brief session in preparation for multifactor authentication. This is the beginning of a secure login flow, even though MFA is covered later.



```

const { email, password, licenseNumber, role } = req.body;

try {
  // Check if email already exists
  if (await User.findOne({ email })) {
    return res.status(409).json({ message: 'Email already exists' });
  }

  // Correct way to check for duplicate license number
  const users = await User.find();
  for (const user of users) {
    const isMatch = await compareField(licenseNumber, user.licenseNumber);
    if (isMatch) {
      return res.status(409).json({ message: 'License number already exists' });
    }
  }

  // Hash password & license number
  const hashedPassword = await hashField(password);
  const hashedLicenseNumber = await hashField(licenseNumber);

  // Generate MFA Secret
  const mfaSecret = authenticator.generateSecret();
  const mfaURI = authenticator.keyuri(email, 'Health-service.click', mfaSecret);
  const qrCode = await qrcode.toDataURL(mfaURI);

  const token = crypto.randomBytes(32).toString('hex');
  const expires = Date.now() + 24 * 60 * 60 * 1000;

  // Create new user (Set `isVerified = true` before saving)
  const user = new User({
    email,
    password: hashedPassword,
    licenseNumber: hashedLicenseNumber,
    role: role.toLowerCase(),
    isVerified: false,
    verificationToken: token,
    verificationTokenExpires: expires, // Set before saving
    mfaEnabled: true,
    mfaSecret, // Store the MFA secret
  });

  // Save user to database (Only once!)
  await user.save();
}

```

Figure 73 Login Function

## Using JWTs to Manage Sessions:

Every protected route uses the `ensureAuthenticated` middleware to verify that the access token is legitimate. This middleware takes the token out of the cookies,

- uses the `verifyToken()` function to confirm it (based on `jsonwebtoken`),
- Adds to `req.user` the decoded user data (role and ID).

Access is instantly refused with a 401 Unauthorised response if the token is lost, expired, or altered. This method preserves a safe, stateless authentication process while doing away with the requirement for serverside session storage.

## Authorisation: RoleBased Access Control (RBAC):

Once a user is authenticated, their role governs which resources they can access. Role enforcement is handled using the custom `authoriseRoles` middleware, which:

- Accepts one or more allowed roles (e.g., doctor, pharmacist)
- Checks if `req.user.role` matches
- Blocks access if unauthorised

### 5.6.1.5 Multifactor Authentication (MFA)

The Secure Prescription System uses Time based One Time Passwords (TOTP) in conjunction with Multifactor Authentication (MFA) to improve security beyond password based login.

Unauthorised access is avoided even if a users login credentials are compromised thanks to this extra verification step.

## MFA Setup During Registration

```
// Generate MFA Secret
const mfaSecret = authenticator.generateSecret();
const mfaURI = authenticator.keyuri(email, 'Health-service.click', mfaSecret);
const qrCode = await qrcode.toDataURL(mfaURI);
```

Figure 74 MFA Variables

The `authenticator.generateSecret()` function of the `otplib` package is used by the backend to create a distinct secret key for every user during user registration. This key is used to generate a TOTP URI, which the `qrcode` library then transforms into a QR code. The client receives the

generated QR code back, enabling the user to scan it with an authenticator app such as Authy or Google Authenticator.

Along with other registration information, the secret is kept in the User models mfaSecret field.

## MFA Flow During Login

The server validates the users email address and password when they log in. If this is the case, the users ID is stored in a temporary cookie (mfa\_session) that is created by the server and expires after five minutes. This cookie indicates that the user is now prepared for MFA after the first factor (password) has been validated.

```
// Store user ID in a temporary session cookie
res.cookie('mfa_session', user._id.toString(), {
  httpOnly: true,
  secure: true,
  sameSite: 'None',
  maxAge: 5 * 60 * 1000, // Expires in 5 minutes
});
```

Figure 75 Session Cookie

The client is then prompted to enter their 6digit code generated by the authenticator app.

## Verifying the TOTP

In the mfaLogin() controller, the server retrieves the users mfaSecret and verifies the onetime code submitted by the user using:

```
const verified = authenticator.verify({ token: totp, secret: user.mfaSecret });
```

Figure 76 Verify TOTP Function

If it is valid, the system issues a secure JWT access token, which is saved in the accessToken httpOnly cookie. To prevent its reuse, the mfa\_session cookie is deleted.

The risk of unwanted access is greatly decreased by this twostep login process, particularly in a medical setting where the application handles private patient and prescription data.

## Security and Compliance Considerations

TOTP is time based and each code is only valid for a brief period of time, it is resilient to replay attacks.

- MFA bypasses are blocked: The login is refused if the session has ended or the TOTP is not accurate.
- The management of sessions is safe: Secure cookies (HTTP Only, Secure, and SameSite=None) are used to store both temporary and permanent tokens.

By using MFA, the application complies with security standards in contemporary web systems and best practices for protecting medical data (OWASP, 2023).

### *5.6.1.6 Field Level Data Encryption*

Securing sensitive data when it's at rest is crucial for any system involving healthcare. The backend uses AES256GCM encryption, which is implemented via a custom utility module using Node.js native crypto library, to protect sensitive data like prescriptions, patient information, and pharmacist instructions.

Mongoose getters and setters make it easy to deploy secure encryption and decryption across several models because this encryption code is abstracted into a reusable utility module (utils/encryption.js).

```

firstName: {
  type: String,
  required: true,
  set: encryptData,
  get: decryptData,
},
lastName: {
  type: String,
  required: true,
  set: encryptData,
  get: decryptData,
},
dateOfBirth: {
  type: String,
  required: true,
  set: encryptData,
  get: decryptData,
},
gender: {
  type: String,
  required: true,
  set: encryptData,
  get: decryptData,
},
phoneNumber: {
  type: String,
  required: true,
  set: encryptData,
  get: decryptData,
},
email: {
  type: String,
  unique: true,
  sparse: true,
  set: encryptData,
  get: decryptData,
},
address: {
  street: { type: String, set: encryptData, get: decryptData },
  city: { type: String, set: encryptData, get: decryptData },
  postalCode: { type: String, set: encryptData, get: decryptData },
  country: {
    type: String,
    required: true,
    set: encryptData,
    get: decryptData,
  },
},
},

```

Figure 77 Patient Model 2

The module uses the AES256GCM algorithm, a secure and efficient symmetric encryption mode that offers both confidentiality and data integrity. The encryption key is securely loaded from an environment variable (ENCRYPTION\_KEY) and parsed as a buffer:

```

const encryptionKey = Buffer.from(process.env.ENCRYPTION_KEY, 'hex');

```

Figure 78 Encryption Key

```

try {
  // Generate a unique IV for each encryption
  const iv = crypto.randomBytes(ivLength);

  // Create cipher instance
  const cipher = crypto.createCipheriv(algorithm, encryptionKey, iv);

  // Encrypt Data
  let encrypted = cipher.update(data, 'utf8', 'hex');
  encrypted += cipher.final('hex');

  // Generate auth tag
  const authTag = cipher.getAuthTag().toString('hex');

  return JSON.stringify({
    iv: iv.toString('hex'),
    authTag,
    encryptData: encrypted,
  });
} catch (error) {
  console.error('Encryption error:', error.message);
  return null;
}

```

Figure 79 Encrypt function

Every time data is encrypted, `crypto.randomBytes()` is used to create a unique Initialisation Vector (IV). By preventing the ciphertext from being reused or altered, the IV and an authentication tag (`authTag`) produced by GCM help to prevent common attacks like replay and padding oracle attacks.

A JSON string with the following structure makes up the encrypted output:

- `iv`: the initialisation vector generated at random
- `authTag`: used to confirm that the encrypted message is intact
- `encryptData`: the data that has been encrypted

Using the Mongoose schema `set()` methods, this is returned and saved straight into the database.

## The Decryption Logic

The `decryptData()` function reverses the process when data is read from the database.

- parses the encrypted JSON that is stored.
- uses the original IV and auth tag to reconstruct the cipher.
- returns the plaintext string after decrypting it.

The developer never has to explicitly decrypt fields because Mongooses `get()` method in schema definitions transparently handles this procedure.

```
try {
  const { iv, authTag, encryptData } = JSON.parse(encryptedString);

  const decipher = crypto.createDecipheriv(
    algorithm,
    encryptionKey,
    Buffer.from(iv, 'hex')
  );

  decipher.setAuthTag(Buffer.from(authTag, 'hex'));

  let decrypted = decipher.update(encryptData, 'hex', 'utf8');
  decrypted += decipher.final('utf8');

  return decrypted;
} catch (error) {
  console.error('Decryption failed:', error.message);
  return null; // Prevent crash
}
```

Figure 80 Decrypt Function

## Why AES256GCM?

- Authenticated encryption: AESGCM protects against tampering by ensuring that data hasn't been changed while in transit or storage.
- Performance: Modern CPUs natively implement AESGCM, which is efficient.
- Consistent and reusable: By centralising the logic into a utility module, code duplication is avoided, and consistent encryption standards are enforced across the codebase.

AESGCM offers authenticated encryption and strong security guarantees, being a NIST approved standard widely used across critical sectors like healthcare and finance (National Institute of Standards and Technology, 2001).

### 5.6.1.7 Realtime Notifications & Email Integration

Realtime notifications and administrative email processes are critical for ensuring responsiveness, reduced delays, and improving the user experience in a prescription system where doctors and pharmacists must communicate quickly. To accomplish this, the projects backend implementation combines email processes with Nodemailer and WebSocket based realtime communication with Socket.io.

#### RealTime Notifications with Socket.io

The backend server establishes a Socket.io connection via index.js, enabling bidirectional communication between connected users. When a user connects, their `userId` is mapped to their socket session using a global `connectedUsers` map. This enables the backend to directly push updates or alerts to a specific user by their ID.

```
// Initialise socket.io
export const io = new Server(server, {
  cors: {
    origin: allowedOrigins,
    credentials: true,
    allowedHeaders: ['Content-Type', 'Authorization', 'Origin', 'Accept']
  },
});

// Store connected users: Map of userId => socket.id
export const connectedUsers = new Map();

// Socket.io connection
io.on('connection', (socket) => {
  console.log(`Client connected: ${socket.id}`);

  socket.on('register', (userId) => {
    connectedUsers.set(userId, socket.id);
    console.log(`Registered user ${userId}`);
  });

  socket.on('disconnect', () => {
    for (const [userId, id] of connectedUsers.entries()) {
      if (id === socket.id) {
        connectedUsers.delete(userId);
        console.log(`User ${userId} disconnected`);
        break;
      }
    }
  });
});
```

Figure 81 Socket.io Connection

When a pharmacist updates the prescription status (e.g., to “Processed” or “Completed”), the system identifies the relevant doctor and emits a `prescriptionupdated` event:

```
// Notify doctor via socket
const doctorSocketId = connectedUsers.get(prescription.doctorId.toString());
if (doctorSocketId) {
  const patient = await Patient.findById(prescription.patientId);

  io.to(doctorSocketId).emit('prescription-updated', {
    prescriptionId: prescription._id,
    status: prescription.status,
    notes: prescription.notes,
    updatedAt: prescription.updatedAt,
    message: 'Prescription updated by pharmacist.',
    patient: patient ? `${patient.firstName} ${patient.lastName}` : 'Unknown patient',
  });
}
```

Figure 82 Doctor Notification



### Benefits:

- Allows for immediate notifications when new or updated prescriptions are made.
- Minimises the need for manual refreshes or polling.
- Enhances collaboration between pharmacists and doctors.

Socket based notification systems are widely used in healthtech and fintech platforms to minimise delays and ensure immediate feedback loops (Socket.io, n.d.).

## Email Verification with Nodemailer

To manage user registrations and enforce administrative approval, the system includes a verification email flow powered by Nodemailer. When a new user registers, the system generates a unique token and expiration timestamp. An email is sent to the admin containing a secure verification link, like:

<https://api.healthservice.click/api/admin/verify/:userId/:token>

This is handled by the `sendEmail()` function in the `config/email.js` file as seen in in figure 83.

Once the admin clicks the link, the backend marks the user as `isVerified = true` if the token is valid and has not expired.

```
export const sendEmail = async (email, role, userId, token) => {
  console.log(process.env.BACKEND_URL);
  const verificationLink = `${process.env.BACKEND_URL}/api/admin/verify/${userId}/${token}`;
  console.log('Generated Verification Link:', verificationLink);

  await transporter.sendMail({
    from: process.env.ADMIN_EMAIL,
    to: process.env.ADMIN_EMAIL, // Admin receives the email
    subject: 'New User Registration - Verify User',
    html: `
      <div style="font-family: Arial, sans-serif; text-align: center; padding: 20px;">
        <h1 style="color: #000;">A new user has registered to Health-Service.click</h1>
        <p style="font-weight: bold; font-size: 16px;">Email: ${email}</p>
        <p style="font-weight: bold; font-size: 16px;">Role: ${role}</p>
        <p style="font-weight: bold; font-size: 16px;">Click below to verify:</p>
        <a href="${verificationLink}" style="
          display: inline-block;
          padding: 12px 24px;
          margin-top: 10px;
          font-size: 16px;
          font-weight: bold;
          color: #fff;
          background-color: #000;
          text-decoration: none;
          border-radius: 5px;">
          Verify User
        </a>
      </div>
    `,
  });
};
```

Figure 83 Send Email Function

Together, this dual system of realtime alerts and email based workflow control strengthens both user interactivity and administrative oversight. These features are particularly vital in medical software, where secure, timely, and traceable communication can have a real impact on patient safety and system reliability.

#### *5.6.1.8 Data Seeding with HPRA verified Medication API*

Put in place a data seeding process that imports authorised medications from a reliable source and fills the medications model database for testing and development in order to guarantee that the Secure Prescription System is based on structured, authentic, and healthcare compliant data.

## Importing HPRA Verified Medications

Used an open data XML file provided by the Health Products Regulatory Authority (HPRA) through Irelands data.gov.ie website as the primary source of data [HPRA, 2024]. A complete list of approved or transfer pending medicinal products for human use is included in this collection. To integrate the data, downloaded the XML file and placed it inside the medicationData/ folder. Wrote a script using xml2js to parser and extract the XML data into usable JSON. Each product was transformed to extract the following fields:

- ProductName (stored as name)
- Active Substances
- Drug IDPK (used as the products unique ID)
- ATC codes and RouteOfAdministration

```

await connectDB();
const xml = await fs.readFile(
  path.resolve(__dirname, '../medicationData/latestHumanlist.xml'),
  'utf-8'
);
const parser = new xml2js.Parser({ explicitArray: false });
const result = await parser.parseStringPromise(xml);

let records = result?.Products?.Product;

if (!records) {
  console.log('No Product data found in XML.');
  return;
}

if (!Array.isArray(records)) {
  records = [records];
}

const medications = records
  .filter((med) => med.ProductName && med.DrugIDPK)
  .map((med) => ({
    name: med.ProductName,
    activeSubstance: Array.isArray(med.ActiveSubstances?.ActiveSubstance)
      ? med.ActiveSubstances.ActiveSubstance.join(', ')
      : med.ActiveSubstances?.ActiveSubstance || null,
    authorisationNumber: med.LicenceNumber,
    atcCode: Array.isArray(med.ATCs?.ATC) ? med.ATCs.ATC.join(', ') : med.ATCs?.ATC || null,
    routeOfAdministration: Array.isArray(med.RoutesOfAdministration?.RoutesOfAdministration)
      ? med.RoutesOfAdministration.RoutesOfAdministration.join(', ')
      : med.RoutesOfAdministration?.RoutesOfAdministration || null,
    productId: med.DrugIDPK,
  }));

await Medication.insertMany(medications);

```

*Figure 84 Medication Seeding*

Once transformed, this data was saved to the MongoDB collection via the Medication model using insertMany(). This approach ensures that all medications available to doctors and pharmacists during development are real, traceable products approved by national healthcare authorities.

## 5.6.2 Frontend

### 5.6.2.1 Project Structure and Initial Setup

The frontend of the project comprises of Vite, a fast modern build tool optimised for performance. The tech stack includes React for building the user interface, TypeScript for static typing, shadcn/ui, Tanstack query for efficient data fetching and Tanstack router for client-side routing.

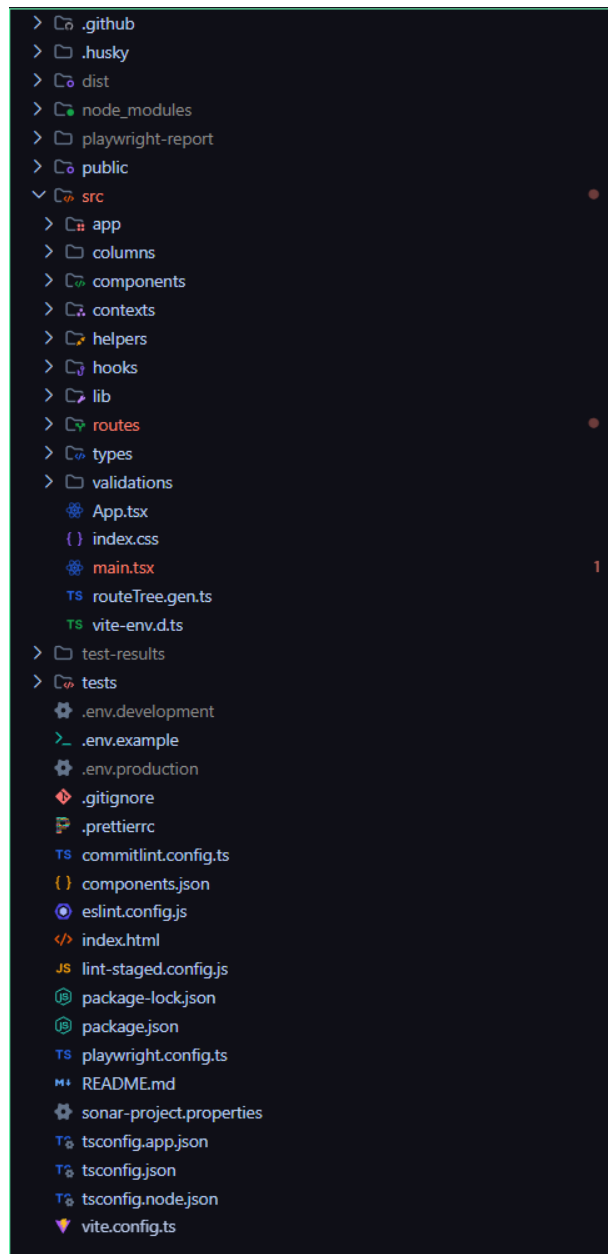


Figure 85 Frontend Project Structure

## Project Structure

To simplify code and separate concerns, the project uses modular and scalable folders.

Important folders and files:

- **src/**: Main source folder containing the application's logic.
  - **app/**: Contains global providers and layout-related configuration.
  - **components/**: Reusable UI components.
  - **columns/**: Configuration for data tables (e.g., patient lists, prescriptions).
  - **contexts/**: React context providers for shared state.

- **helpers/**: Utility functions used across the app.
- **hooks/**: Custom React hooks.
- **lib/**: Shared libraries such as API clients.
- **routes/**: Route definitions and page components, aligned with TanStack Router.
- **types/**: TypeScript type definitions and interfaces.
- **validations/**: Zod schemas for form validation.
- **App.tsx**: The root component where the app is initialized.
- **main.tsx**: Entry point where the app is mounted to the DOM.
- **public/**: Static files like favicon and index.html.
- **tests/**: End-to-end and unit tests, including Playwright test results.
- **vite-env.d.ts**: Type definitions for Vite's environment variables.

## Code Quality

- **Prettier**: .prettierrc formats code automatically.
- **ESLint**: Custom setup (eslint.config.js) ensures code meets linting standards and conventions.
- **Husky**: Git hook manager in .husky/. Linting and formatting are automated via pre-commit hooks.
- **Lint-Staged**: Improved commit efficiency by only linting and formatting staged files with Husky.
- **Playwright-report/** stores end-to-end testing findings.
- **Commitlint**: monitors commit messages for consistency and significance.
- **SonarQube Integration**: Sonar-project.properties sets up continuous code quality analysis.

### 5.6.2.2 Component based Architecture

PharmaLink's frontend uses React's component-based architecture. Modularity improves reusability, maintainability, and concern separation. Each component handles its own logic and presentation, making scaling and updating straightforward.

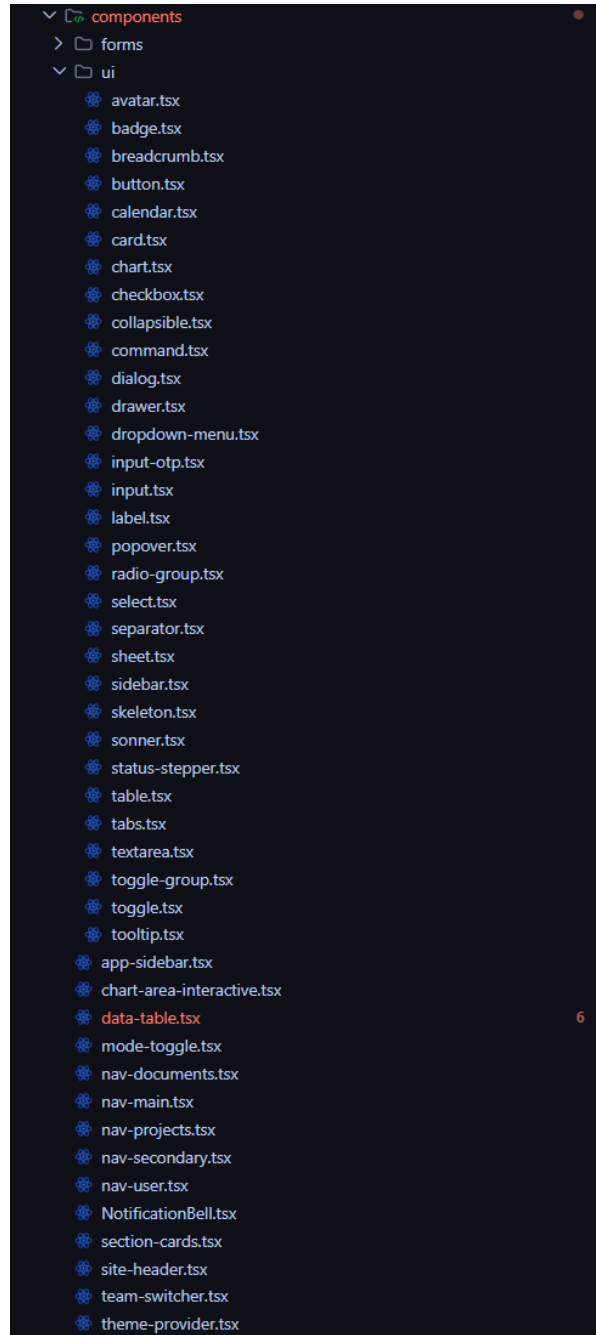


Figure 86 UI Components

## UI Components and Shared Structure

The components/ directory has subfolders for ui/, forms/, and navigation (nav-main.tsx, nav-user.tsx). These contained buttons, inputs, modals, dropdowns, tooltips, and layout containers.

Most UI primitives use the shadcn/ui component system, which includes Radix UI for accessibility and allows Tailwind CSS changes. The app uses reusable components such as `<Button />`, `<Dialog />`, `<Select />`, and `<Table />` to maintain a consistent design language.

## Layout Architecture and Sidebar Navigation

PharmaLink's persistent layout has a responsive sidebar for doctors and pharmacists. The layout in `__root.tsx` conditionally renders navigation based on the current route:

The sidebar itself (`app-sidebar.tsx`) is composed using structured layout primitives (`SidebarHeader`, `SidebarContent`, `SidebarFooter`) and integrates contextual role-checks via `AuthContext`:

```
{user?.role === 'doctor' && <NavMain items={doctorNav} />}  
  {user?.role === 'pharmacist' && <NavMain items={pharmacistNav} />}
```

*Figure 87 Conditional Rendering*

This modular architecture shows pharmacists and doctors only workflow-relevant UI. Doctors are also conditionally offered a Quick Prescription button for fast prescription creation.

### Data table component

PharmaLink's robust `DataTable` component, designed with TanStack Table v8 is reusable. This component powers most tabular views:

- Prescription lists
- Appointment schedules
- Patient directories

### Features Include:

- Sorting, pagination, filtering.
- Select row, edit and delete inline.
- Mobile-responsive layout and controls.

```

{userRole === 'doctor' && selectedId && (
  <>
    {editUrl && (
      <Button
        variant="default"
        size="sm"
        onClick={() => navigate({ to: editUrl(selectedId) })}
        className="mw-full sm:w-auto font-semibold cursor-pointer px-8"
      >
        Edit
      </Button>
    )}
    {onDelete && (
      <Button
        variant="destructive"
        size="sm"
        onClick={() => onDelete(selectedId)}
        className="mw-full sm:w-auto font-semibold cursor-pointer px-8"
      >
        Delete
      </Button>
    )}
  </>
)}
)

```

Figure 88 Inline Row Actions

### 5.6.2.3 Routing and Navigation

The modern, file-based routing system at PharmaLink uses TanStack Router, a type-safe and versatile React routing solution. Separated concerns, role-based access control, and deep linking for prescriptions, patients, and appointments are built into the routing architecture.

## Routing Structure

The `src/routes/` directory contains frontend routing configuration. The folder structure matches the application URL, improving maintainability and developer experience.

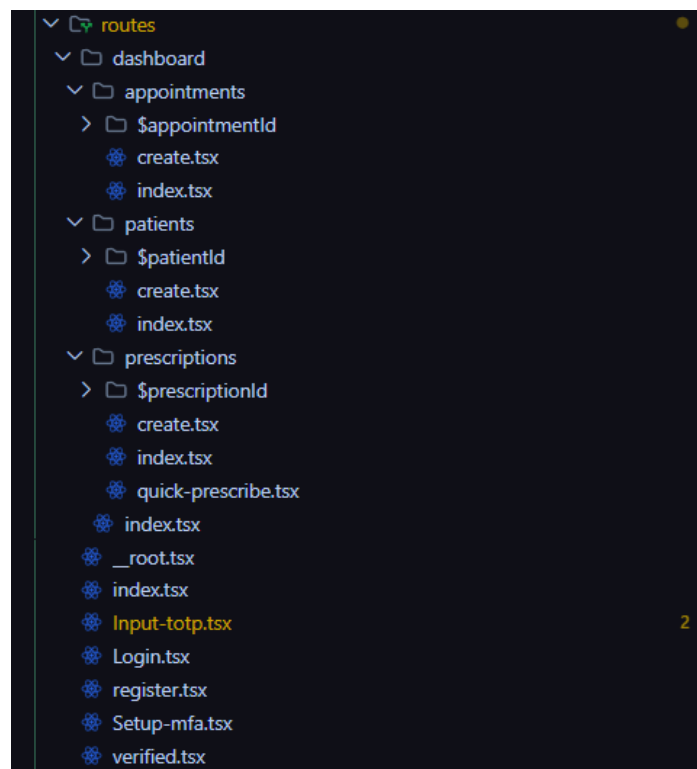


Figure 89 Routes



This structure leverages dynamic route segments (e.g., \$prescriptionId) to handle CRUD operations for specific resources.

## Root Layout and Conditional Rendering

\_\_root.tsx is the application's top layout. It renders child routes using TanStack Router's Outlet component and dynamically decides whether to present the dashboard layout based on the URL.

```
// Check if the current route starts with /dashboard
const isDashboardRoute = matchRoute({ to: '/dashboard', fuzzy: true });

return (
  <div className="min-h-screen bg-gray-100 dark:bg-black text-black dark:text-white">
    {isDashboardRoute ? (
      <SidebarProvider>
        <AppSidebar variant="inset" />
        <SidebarInset>
          <SiteHeader />
          <main className="flex-1">
            <Outlet />
          </main>
        </SidebarInset>
      </SidebarProvider>
    ) : (
      <>
        <div className="flex justify-end p-5">
          <ModeToggle />
        </div>
        <Outlet />
      </>
    )}
  </div>
  <Toaster />
  <TanStackRouterDevtools />
</div>
);
```

Figure 90 \_\_root Layout

If the user is in the /dashboard route it will render

- AppSidebar: role-based navigation sidebar
- SiteHeader: header with controls for the user
- SidebarInset: Main content container for the dashboard routes

Or else it will render the public pages like Login, Register, and MFA Setup with a simpler layout.

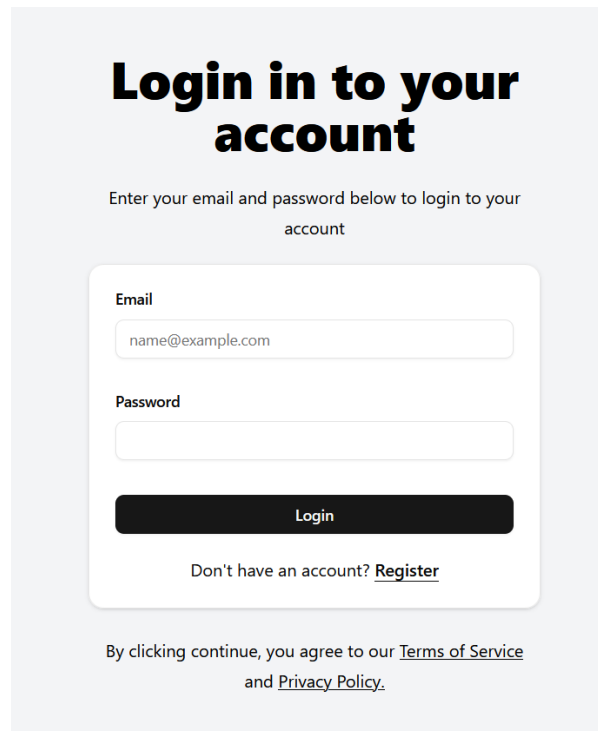
A public login page with a light gray background. At the top, the text "Login in to your account" is displayed in a large, bold, black font. Below this, a smaller line of text says "Enter your email and password below to login to your account". The main form is a white rounded rectangle containing two input fields: "Email" with the placeholder text "name@example.com" and "Password". Below the password field is a black "Login" button. Under the button, there is a link: "Don't have an account? [Register](#)". At the bottom of the form, there is a line of text: "By clicking continue, you agree to our [Terms of Service](#) and [Privacy Policy](#)."

Figure 91 Public Login Page

## Dashboard Overview Page

The dashboard home page (/dashboard/index.tsx) serves as the main landing page for authenticated users. It is highly dynamic and role aware.

- Fetches all prescriptions using a custom hook usePrescriptions powered by TanStack Query for server state management.
- Filters prescriptions based on the logged-in user's role (Doctor or Pharmacist) using AuthContext.
- Renders a reusable DataTable component with column sorting, filtering by patient name, row selection, and row action buttons for prescriptions.
- Deletes prescriptions with optimistic UI feedback using useMutation.

```
const { user } = useContext(AuthContext);
const queryClient = useQueryClient();
const { data: prescriptions, isLoading, isError } = usePrescriptions();

const deleteMutation = useMutation({
  mutationFn: deletePrescription,
  onSuccess: () => {
    toast.success('Prescription deleted successfully');
    queryClient.invalidateQueries({ queryKey: ['prescriptions'] });
  },
  onError: () => toast.error('Failed to delete Prescription'),
});
```

#### 5.6.2.4 Form Handling and Validation

Form handling is crucial to PharmaLink, especially for prescription creation and editing. The frontend uses two strong libraries to provide a smooth, user-friendly, and robust form experience:

- Manage form state with react-hook-form.
- Zod for schema-based validation.

This solution manages complicated forms with dynamic fields at high scale with good performance and validation accuracy.

### Form Management with React Hook Form:

React Hook Form is utilized widely throughout the application to efficiently handle form state with minimal re-renders. It offers:

- Integration with controlled components is simple.
- Built-in error handling.
- Enhanced performance for large or dynamic forms.

```
const {
  register,
  control,
  handleSubmit,
  reset,
  formState: { errors },
} = useForm<PrescriptionFormData>({
  resolver: zodResolver(prescriptionSchema),
  defaultValues: prefill || {
    patientId: '',
    pharmacistId: '',
    notes: '',
    items: [
      {
        medicationId: '',
        specificInstructions: '',
        dosage: '',
        amount: '',
        repeats: 1,
      },
    ],
  },
});
```

Figure 93 React Hook Form Usage

Features include:

- Field Registration.
- Controlled Inputs for custom components like Select.

- Dynamic fields using useFieldArray for adding/removing prescription items.

## Schema Validation using Zod

Zod defines strict form data schemas to check user input against established rules.

Prescription item validation example:

```
const itemSchema = z.object({
  medicationId: z
    .string()
    .nonempty({ message: 'Medication is required.' })
    .regex(objectIdRegex, { message: 'Invalid medication format.' }),

  specificInstructions: z
    .string()
    .nonempty({ message: 'Specific instructions are required.' }),

  dosage: z.string().nonempty({ message: 'Dosage is required.' }),

  amount: z.string().nonempty({ message: 'Amount is required.' }),

  repeats: z.number().min(0, 'Repeats cannot be negative'),
});
```

Figure 94 Zod Validation Schema

## Dynamic Prescription Form Design

The CreatePrescriptionForm is a reusable form component that allows doctors to create prescriptions. Its key features include:

- Patient and pharmacist selection with the Select component with controlled values
- Dynamic prescription items with useFieldArray for adding/removing items
- AsyncMedicationSearch component for fetching medications dynamically
- Error handling using inline error messages via the errors from react hook form.

## AsyncMedicationSelect Component

- Allow users to search for medications from the backend using an async API call
- Integrated react-hook-form using Controller for form state management
- Debounced search to prevent API spamming
- Only starts searching when 2+ characters have been entered
- Uses custom useMedicationSearch hook to fetch medication from the backend
- Automatically shows pre-filled selected medication which is useful for the edit prescription form or duplicate prescription

```

export function AsyncMedicationSelect({ field, initialValue }: Props) {
  const [query, setQuery] = useState('');
  const [hasInteracted, setHasInteracted] = useState(false);
  const [debounced] = useDebounce(query, 300);

  const { data = [], isLoading } = useMedicationSearch(
    debounced.length >= 2 ? debounced : ''
  );

  const [selectedMedication, setSelectedMedication] =
    useState<Medication | null>(initialValue ?? null);

  useEffect(() => {
    if (initialValue && field.value === initialValue._id) {
      setSelectedMedication(initialValue);
      return;
    }

    if (!selectedMedication && field.value) {
      const match = data.find((m) => m._id === field.value);
      if (match) setSelectedMedication(match);
    }
  }, [data, field.value, initialValue]);

  // Dynamically show label in the input if user hasn't typed yet
  const inputDisplayValue =
    hasInteracted || query.length > 0 ? query : selectedMedication?.name || '';

  return (
    <div className="space-y-2">
      <Command className="rounded-md border shadow-sm">
        <CommandInput
          value={inputDisplayValue}
          onValueChange={(val) => {
            setQuery(val);
            if (!hasInteracted) setHasInteracted(true);
          }}
          placeholder="Search medications..."
          className="placeholder:text-muted-foreground"
        />
        <CommandList>
          {isLoading && <CommandItem disabled>Loading...</CommandItem>}

          {hasInteracted &&
            query.length >= 2 &&
            !isLoading &&
            data.length === 0 && (
              <CommandEmpty>No medications found.</CommandEmpty>
            )}

          {data.map((med) => (
            <CommandItem
              key={med._id}
              value={med.name}
              onSelect={() => {
                field.onChange(med._id);
                setSelectedMedication(med);
                setQuery('');
                setHasInteracted(false); // Reset for display purposes
              }}
            >
              {med.name}
            </CommandItem>
          ))}
        </CommandList>
      </Command>

      {selectedMedication && (
        <div className="text-md font-medium">
          Selected: { ' ' }
          <span className="font-semibold">{selectedMedication.name}</span>
        </div>
      )}
    </div>
  );
}

```

Figure 95 Async Medication Select

#### 5.6.2.5 State Management and Data Fetching

PharmaLink manages state and retrieves data using:

- TanStack React Query
- React Context API
- Built-in React hooks: useState, useEffect, useContext

This architecture enforces user role-based data access and keeps the frontend reactive, efficient, and in sync with backend data.

## React Query – Server State Management

PharmaLink manages server state across the app with React Query. React Query offers:

- Caching: Reduces unnecessary API calls.
- Background Refetching: Keeps data fresh automatically.
- Mutation handling: Easily handle create, delete and edit operations
- Query Invalidation: Automatically refetch data after a mutation

### UsePrescriptions Hook:

```
export const usePrescriptions = () => {  
  const { user } = useContext(AuthContext);  
  
  return useQuery({  
    queryKey: ['prescriptions'],  
    queryFn: getPrescriptions,  
    enabled: !!user,  
  });  
};
```

Figure 96 usePrescriptions Hook

- Fetches all the prescriptions from the backend
- Query is only enabled if the user is logged in
- Data is cached dynamically
- The query can be invalidated using `queryClient.invalidateQueries` to refresh data after changes such as create or edit

## Mutations – Creating, Editing and Deleting Data

For data mutations, useMutation is used:

```
const deleteMutation = useMutation({
  mutationFn: deletePrescription,
  onSuccess: () => {
    toast.success('Prescription deleted successfully');
    queryClient.invalidateQueries({ queryKey: ['prescriptions'] });
  },
  onError: () => toast.error('Failed to delete Prescription'),
});
```

Figure 97 useMutation

- Call API function – deletePrescription
- On success: show a notification and invalidate the prescriptions query so React Query refetches the latest data.
- On error: Show an error toast

## API Layer – Axios Integration

Api.ts defines all API functions utilizing Axios:

- A central baseUrl and withCredentials configuration.
- Use handleApiError() for consistent logging.

A sample API function:

```
// Verify OTP
export const verifyOTP = async (otp: string) => {
  try {
    const res = await api.post('/auth/login/mfa', { totp: otp });
    return res.data;
  } catch (error) {
    handleApiError(error);
  }
};
```

Figure 98 OTP API Function

## React Hooks – Local State Management

React's built-in hooks are used for:

- useState: Manage local UI state (e.g., search input, form fields).
- useEffect: Handle side effects (e.g., reset form on prefill change, fetch data on mount).

- useContext: Access global user state via AuthContext

#### 5.6.2.6 Authentication and Role based Access Control

PharmaLink enforces a secure and role-aware frontend experience through a custom authentication context and React Query-based user session management. This system ensures only authorized users can access protected routes and features — tailored to either doctors or pharmacists.

## Authentication Flow

Axios withCredentials authenticates API requests using session cookies maintained by the backend.

Frontend user session data is fetched:

```
const {
  data: user,
  isLoading,
  isFetching,
  isError,
  refetch,
} = useQuery({
  queryKey: ['user'],
  queryFn: fetchUser,
  enabled: true,
  retry: false,
  refetchOnWindowFocus: false,
});
```

Figure 99 useQuery Auth

The fetchUser function:

- Sends a GET /auth/me request.
- Returns the currently logged-in user's \_id, email, and role.
- Stores this data in React Query's cache for global access.

## AuthContext – Global User Access

Custom AuthContexts provide user data and session state to the entire application.

```
<AuthContext.Provider
  value={{ user, isLoading: false, refetchUser: refetch }}
>
  {children}
</AuthContext.Provider>
```

Figure 100 AuthContext

- Wraps the entire application in main.tsx.
- Blocks the app rendering with a loading screen while the user session is being fetched



## useAuth Hook

```
export const useAuth = () => {  
  const context = useContext(AuthContext);  
  if (!context) throw new Error('useAuth must be used within an AuthProvider');  
  return context;  
};
```

Figure 101 useAuth Hook

Allows components to easily access user data:

```
const { user } = useAuth();
```

Figure 102 useAuth Usage

## Role Based Access Control

The user's role (doctor or pharmacist) is used throughout the app to control:

- Sidebar Navigation: renders different menu items based on the users role.
- Data filtering: filters prescriptions by doctor or pharmacist id.
- Page Access: Protected routes if the user is authenticated.
- Quick Prescription button: only shows if the role is doctor.

```
{user?.role === 'doctor' && (  
  <SidebarMenu>  
    <SidebarMenuItem className="flex items-center gap-2">  
      <SidebarMenuButton  
        tooltip="Quick Create"  
        className=  
          "min-w-4 w-3/4 bg-primary text-primary-foreground duration-200 ease-linear hover:bg-primary/90 hover:text-primary-foreground active:bg-primary/90 active:text-primary-foreground mx-3 cursor-pointer"  
      >  
        <PlusCircleIcon />  
        <Link to="/dashboard/prescriptions/create">  
          { ' ' }  
          <span className="font-semibold">Quick Prescription</span>  
        </Link>  
      </SidebarMenuButton>  
    </SidebarMenuItem>  
  </SidebarMenu>  
)}
```

Figure 103 Quick Prescription Button

## Summary of Authentication System

- AuthProvider: fetches the user session and provides a global context
- useAuth: A custom hook to access user and session utilities.
- AuthContext: Stores the user, isLoading, and refetchUser.
- fetchUser API: Fetches the user data from secure backend.
- React Query: Manages the user session caching and loading state.
- Role checks: Used in the UI and route logic to enforce access control.

This authentication and role-based access system ensures that only verified users can use PharmaLink — and that their experience is personalized based on their medical role. It also establishes a secure foundation for protecting sensitive healthcare data on the frontend.

#### 5.6.2.7 RealTime Notifications

PharmaLink uses Socket.IO for real-time prescription updates for doctors and pharmacists. Clinical process efficiency and responsiveness are improved by this.

## Overview of Architecture

The real-time notification system is built with the following key pieces:

- Socket.ts: Initialises and configures the Socket.IO client.
- SocketProvider: Manages socket connection and event subscriptions.
- useSocket: Custom hook to access the socket context.
- NotificationBell: UI component showing unread notifications.

## Socket Initialisation

```
const socket = io(BASE_API_URL, {
  withCredentials: true,
  autoConnect: false,
  transports: ['websocket'],
});

export default socket;
```

Figure 104 Socket Initialisation

- The socket is configured to connect using WebSocket transport.
- Credentials (cookies) are sent for secure session validation.
- Connection is manually initiated when a user is authenticated.

## Socket Provider – Connection Lifecycle

Socket Provider is a global context that connects the socket once a user is available from the authContext. It will register the user on the server using `socket.emit('register', user._id)`.

Subscribes to events based on the user's role such as pharmacists listening for new-prescriptions and doctors listening for prescription-updated.

```

if (user.role === 'pharmacist') {
  socket.on('new-prescription', handleNewPrescription);
}

if (user.role === 'doctor') {
  socket.on('prescription-updated', handlePrescriptionUpdated);
}

```

Figure 105 Socket User Roles

These events update the local notifications state. unreadCount is incremented to show the badge in the UI.

```

const [notifications, setNotifications] = useState([]);
const [unreadCount, setUnreadCount] = useState<number>(0);

```

Figure 106 Socket State

```

const handleNewPrescription = (data) => {
  console.log('[socket.io] New prescription assigned:', data);
  setNotifications((prev) => [...prev, { type: 'new', ...data }]);
  setUnreadCount((prev) => prev + 1);
};

```

Figure 107 New Prescription Notification Function

## useSocket – Custom Hook

```

export const useSocket = () => {
  const context = useContext(SocketContext);
  if (!context)
    throw new Error('useSocket must be used within a SocketProvider');
  return context;
};

```

Figure 108 useSocket Hook

Provides safe access to the socket context throughout the app.

## NotificationBell Component

- Displays a red dot when there are unread notifications.
- Opens a popover showing a scrollable list of messages.
- Each notification is clickable and navigates to a relevant prescription.
- Users can "Mark all as read" to clear the unread count.

```

<div className="space-y-2 max-h-60 overflow-y-auto">
  {notifications.length === 0 ? (
    <p className="text-sm text-muted-foreground">
      No new notifications
    </p>
  ) : (
    notifications.map((notification, id) => (
      <div
        key={id}
        onClick={() =>
          navigate({
            to: `/dashboard/prescriptions/${notification.prescriptionId}`,
          })
        }
        className="rounded border px-3 py-2 text-sm cursor-pointer hover:bg-muted transition"
      >
        <div className="font-medium mb-1">
          {notification.type === 'new'
            ? 'New Prescription Assigned'
            : 'Prescription Updated'}
        </div>
        <div className="text-muted-foreground text-xs">
          {notification.patient && (
            <span className="font-medium">
              {notification.message} - {notification.patient}
            </span>
          )}
        </div>
        {notification.createdAt || notification.updatedAt ? (
          <div className="text-muted-foreground text-[11px] mt-1">
            {new Date(
              notification.createdAt || notification.updatedAt
            ).toLocaleString()}
          </div>
        ) : null}
      </div>
    ))
  )}
</div>

```

Figure 109 Displaying Notification

## Summary of Real-Time Notification System

- Real-time updates: Socket.IO with role-based events.
- Secure User connection: Registered with backend after login.
- Global State: Stored inside the SocketProvider.
- Contextual Navigation: Clicking notifications routes to the details.

PharmaLink's real-time architecture lets users react quickly, streamline process, and avoid missing prescription updates while retaining secure, role-based control.

## 6 Testing

### 6.1 Introduction

The software development lifecycle includes testing to ensure that the PharmaLink system works reliably, meets functional requirements, and delivers an intuitive user interface for doctors and pharmacists.

Automated and real user testing were used to validate the system from backend logic to frontend interface in this project.

Used various tools and methodologies such as:

- Playwright: For end-to-end (E2E) testing of the frontend, simulating real user flows such as login, prescription creation, and access control.
- Vitest: Frontend unit testing and component-level testing to ensure separated logic works.
- Jest: For API endpoint, service, and utility function backend unit testing.
- User Testing: Conducted with real users (e.g. pharmacists and doctors) to gather feedback on usability, functionality, and role-based workflows.

This chapter outlines the testing strategy used throughout PharmaLink.

### 6.2 Pharmacist Testing

#### 6.2.1 Pharmacist Feedback

During user testing, a practicing pharmacist provided a detailed assessment of Healthmail and PharmaLink's improvements. Professional perspective supplied real-world context to assess the system's practicality and usability beyond technical performance.

Healthmail has limited search functionality, inconsistent prescription formatting, unclear prescriber details, and disorganized communication procedures, according to the pharmacist.

Case-sensitive search behavior, manual archiving, lack of visibility across systems (e.g., hospitals, private practices), and transcription errors due to vague or misinterpreted dosage instructions all showed an inefficient and error-prone system for pharmacy staff.

PharmaLink immediately addresses several of these issues. The pharmacist praised the system's secure, centralized patient interface, date-ordered prescription view, and clear, traceable prescriber-pharmacist communication. The feedback highlighted how PharmaLink decreased patient misidentification, eliminated manual archive dependence, and quicker prescription validation using improved prescriber notes and organized input fields.

The pharmacist considered a query system to be innovative even in its basic form. A built-in doctor-to-pharmacist communication module would replace email, physical notes, and patient discussions with a single, traceable channel which improves clarity, accountability, and treatment speed.

This response clearly confirmed PharmaLink solves clinical problems. It showed that the system's usability and design match pharmacy operations and suggested ways to improve medical prescribing productivity and safety.

Below in figure 110, is the feedback document provided for the pharmacist.

## Review for Dan Esmonde

14/04/2025

PSI Registration Name: Daniel O'Neill

PSI Registration Number: 13193

Title: Supervising Pharmacist Boots Gorey

### Issues with Healthmail

Issues include:

- Basic Outlook email address interface with registration rule applied that only allows registered medical professionals (medical doctors, registered nurses, dentists), practice surgeries and pharmacies (via Supervising Pharmacist) to create accounts and communicate.
- Increasingly private healthcare services are transitioning to using a Patient Hub... if the pharmacy operates according to legislation... defining criteria for the electronic transmission of prescriptions ... this allows for only Healthmail transfer between clinician and pharmacy hence the patient cannot send their individual prescription to Healthmail using non-Healthmail approved email addresses. This ultimately causes excessive stress to patient and pharmacy teams, multi-channel communication (pharmacy phoning to request prescription), delays in receiving prescription and hence delay to medical treatment until the pharmacy receives either Healthmail copy of prescription or physical prescription in the post.
- Search criteria – operates the same as Outlook search! Places “search hits” at the top. The search is case and spelling sensitive and keywords (eg patient name and address) must be in the subject line or body of the email. If the prescription was sent as an attachment the basic search will not yield desired results. Searches are not in date order and hence older or superseded prescriptions can be shown as top hits and increases potential to dispensing errors.
- Archiving emails manually – emails are not separated into patient folders ... rather in read receipt format archived manually by pharmacy staff once processed to folders sorted by Month and Year archive. Therefore, searching for historical prescriptions can be difficult, time consuming and yield inaccurate results (eg showing patients with different names eg same forename but differing surnames).
- Layout template for prescriptions varies depending on surgery.
- Not all prescribers adopt the same template – most have adopted a relatively uniform layout (to meet all legal requirements of prescriptions) but some have issues eg omission of key prescription requirements.
- Professionally I have encountered cases where there are notable differences between GP view and pharmacy view... ie what surgery can see and what was transferred differ.
- Issues pertaining doctors' instruction interpretation eg use of dose codes and free type and hence can lead to dispensing errors  
eg 10 x Amoxicillin 500mg ONE TO BE TAKEN THREE TIMES DAILY (pre-programmed dose code). One to be taken twice daily as directed. FOR FIVE DAYS.  
Above we see GP consultation notes would state patient prescribed 500mg Amoxocillin twice daily for 5 days. There is a transcription error and leads to therapeutic delay
- GMS numbers of doctors often omitted off Healthmail prescriptions – important for PCRS medicine reimbursement claims.

## Review for Dan Esmonde

14/04/2025

- Prescriber details may not always be clear.
- Hospitals do not routinely have access!
- Healthmail is still not universal with all GP surgeries. Interfaces such as Healthlink integrate with software systems, but Healthlink has no adaptation to Community Pharmacies.
- Large number of Dental Surgeries and Psychiatric Services do not use Healthmail or Healthlink nor do they have access.

Figure 110 Pharmacist Feedback

## 6.2.2 Analysis & Future Improvements

PharmaLink is a secure, user-centered prescription management solution that solves several of Healthmail's problems. The platform improves security, usability, and prescription workflow efficiency by systematic requirements gathering, deliberate system design, and broad user validation.

An experienced pharmacist confirmed that PharmaLink overcomes major operational concerns such as patient misidentification, ineffective archiving, confusing prescriber instructions, and slow, fragmented communication. The platform's unified patient dashboard, secure communication paths, mandatory field validation, and auditability improve current processes. Further enhancements may increase PharmaLink's value and impact. A secure, traceable doctor-to-pharmacist query process would be a major improvement. Pharmacists could request prescription clarifications such as dosage instructions, medication clarification, and administrative details directly from the platform without using email or human interventions. This would improve professional communication, prescription filling, and healthcare team administration.

Other improvements may include:

- **Prescription Analytics Dashboard:** Showing doctors and pharmacists prescribing trends, common errors, and patient drug histories should improve clinical decision-making.
- **Extension of User Roles:** Adding administrative staff for non-clinical prescription management could improve flexibility for larger healthcare providers.
- **Mobile Access:** Creating a PharmaLink app for doctors and pharmacists to handle prescriptions on the go.
- **Patient Notifications:** Notifying patients when a prescription is dispatched, queried, or ready for collection could improve transparency and minimize pharmacy workload.

Below in figure 111, is the problems solved by the PharmaLink application as per the pharmacist.



## Issues solved by Pharmalink

Issues solved include:

- Secure hub with defined registration criteria.
- Patient-hub presentation of Pharmalink – unlike Healthmail it is not based on email receipt ordered by most recent received to Pharmacy.
- All patient data in one secure location and for pharmacist conducting clinical and legal screen of prescription having all data on one source is amazing.
- Reduces risk of wrong patient selection.
- Direct communication between prescriber and pharmacy. Prescriber notes clearer. Option to extend to add pharmacy query section (this would document all professional communication and gives rise to faster response rate with better medical outcomes).
- Obvious tracking of prescription status. Solves issues with wrong information being provided to patients re-prescription processing and unrealistic wait times.
- Prescriptions sorted via date order in patients account.
- Direct traceable communication line between prescriber and pharmacy. Takes away need for long, detailed patient record notes detailing professional intervention .... At present queries must be sent back to prescriber by email (one interface), document intervention on patients dispensing record (second interface), annotate physically onto prescription (interface 3), verbally speak with patient (interface 4 – no physical record) and ensure responses are monitored (will only come in as basic email) and thus reliance on support staff to monitor and update notes and can involve pharmacist or GP not directly involved in query – Pharmalink has a documented layout and removes this tedious administrative burden.

*Figure 111 Pharmacist Issues Solved*

## 6.3 Front End Testing

Frontend testing verifies that a web application's user interface (UI) works properly, looks well, and provides a consistent experience across devices and browsers. In BrowserStack (2023), "frontend testing ensures that all parts of the user interface work properly and that the application provides a seamless experience for users, from functionality to responsiveness and visual consistency".

Frontend testing was necessary for doctors and pharmacists to use PharmaLink efficiently and intuitively. Two main frontend testing methods were used:

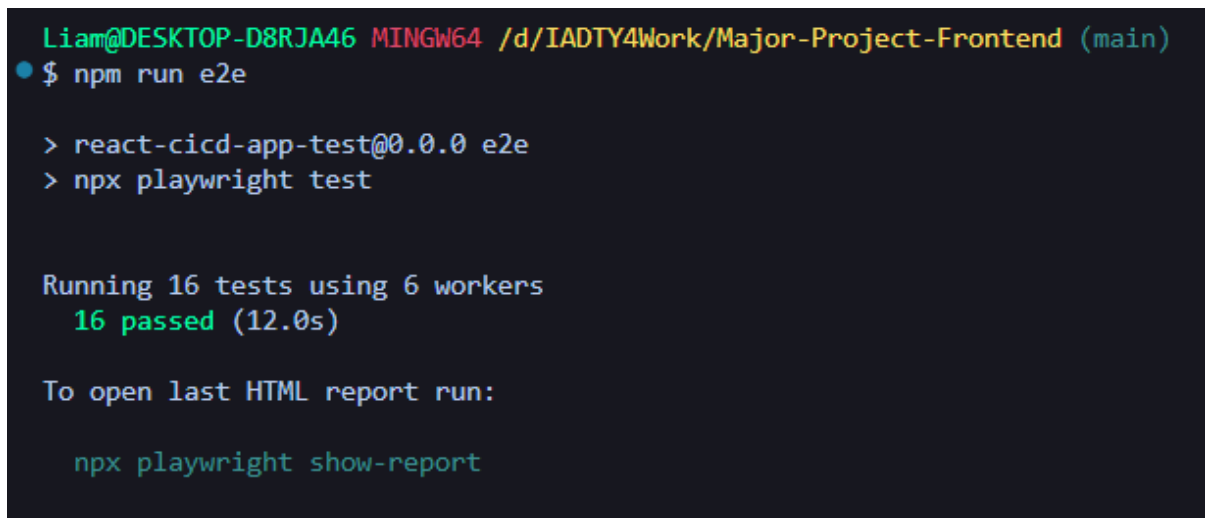
- E2E Testing: Logging in, writing a prescription, and confirming role-based access restriction were automated with Playwright. E2E testing validated frontend and backend procedures to ensure end-user behaviour.
- Unit Testing: Vitest was used for isolated component testing. Verifying that form inputs, buttons, and modals worked independently was required. Unit tests provided fast frontend logic feedback without application installations.

### 6.3.1 End-to-End Testing

E2E testing checks an application's workflow from the user's perspective. Microsoft (2023) states that E2E tests imitate real-world usage to ensure that "all pieces of an application work together properly in real-life scenarios" across many components and layers.

Each E2E test mimicked real user experiences to ensure authentication, dashboard navigation, and prescription management worked. Tests showed that pharmacists could only see their prescriptions, whereas doctors could see all patient and prescription records.

Below in figure 101, a screen capture of the end-to-end tests can be seen passing

A terminal window with a dark background and light-colored text. The prompt is 'Liam@DESKTOP-D8RJA46 MINGW64 /d/IADTY4Work/Major-Project-Frontend (main)'. The user enters '\$ npm run e2e'. The output shows the command being run, followed by 'Running 16 tests using 6 workers' and '16 passed (12.0s)'. A message suggests opening the HTML report with 'npx playwright show-report'.

```
Liam@DESKTOP-D8RJA46 MINGW64 /d/IADTY4Work/Major-Project-Frontend (main)
$ npm run e2e

> react-cicd-app-test@0.0.0 e2e
> npx playwright test

Running 16 tests using 6 workers
  16 passed (12.0s)

To open last HTML report run:

npx playwright show-report
```

Figure 112 E2E Tests Passing

Below in figure 111, is an image of the tests passing for both firefox and chrome browsers:

Q

All 16Passed 16Failed 0Flaky 0Skipped 0

26/4/2025, 17:02:16Total time: 12.0s

▼ dashboard/dashboard.spec.ts

✓ Dashboard page › should load dashboard and show welcome messagechromium1.5s

dashboard/dashboard.spec.ts:6

✓ Dashboard page › should allow navigating to patients pagechromium1.6s

dashboard/dashboard.spec.ts:13

✓ Dashboard page › should allow navigating to prescriptions pagechromium1.6s

dashboard/dashboard.spec.ts:20

✓ Dashboard page › should allow navigating to appointments pagechromium1.6s

dashboard/dashboard.spec.ts:27

✓ Quick Prescription button › should navigate to Quick Prescription pagechromium1.5s

dashboard/dashboard.spec.ts:38

✓ Dashboard page › should load dashboard and show welcome messagefirefox1.7s

dashboard/dashboard.spec.ts:6

✓ Dashboard page › should allow navigating to patients pagefirefox2.0s

dashboard/dashboard.spec.ts:13

✓ Dashboard page › should allow navigating to prescriptions pagefirefox2.0s

dashboard/dashboard.spec.ts:20

✓ Dashboard page › should allow navigating to appointments pagefirefox2.1s

dashboard/dashboard.spec.ts:27

✓ Quick Prescription button › should navigate to Quick Prescription pagefirefox1.0s

dashboard/dashboard.spec.ts:38

▼ prescriptions/prescription.spec.ts

✓ Prescriptions page › should load the prescriptions pagechromium1.4s

prescriptions/prescription.spec.ts:6

✓ Prescriptions page › should show "Add Prescription" button for doctorschromium5.0s

prescriptions/prescription.spec.ts:13

✓ Prescriptions page › should search for a patient by namechromium5.0s

prescriptions/prescription.spec.ts:25

✓ Prescriptions page › should load the prescriptions pagefirefox766ms

prescriptions/prescription.spec.ts:6

✓ Prescriptions page › should show "Add Prescription" button for doctorsfirefox1.9s

prescriptions/prescription.spec.ts:13

✓ Prescriptions page › should search for a patient by namefirefox3.9s

prescriptions/prescription.spec.ts:25

Figure 113 Firefox & Chrome Tests

Below is an example of end-to-end test code in figure 112:

```
test('should allow navigating to patients page', async ({ page }) => {  
  await page.goto('/dashboard');  
  await page.getByRole('link', { name: /Patients/i }).click();  
  await expect(page).toHaveURL(/\/patients/);  
  await expect(page.getByText(/Patients/i)).toBeVisible();  
});
```

Figure 114 Navigation Test

## Challenges Managing Authentication Credentials

Secure authentication for automated testing was a major difficulty during End-to-End testing. PharmaLink uses two-factor authentication (TOTP) in addition to email and password access, complicating login automation.

Using environment variables, a dedicated test user's email and password were safely loaded from a .env.development file. TOTP input automation was not possible within the project timeline.

During the first login, the user must manually enter the TOTP code within a 30-second timeout. To simplify testing, Playwright's storageState feature saved the authenticated session (cookies and localStorage) to a storageState.json file.

Reusing this cached information in subsequent E2E tests bypassed login and improved test speed and reliability.

The testing process focused on securely managing environment variables and preventing version control credential disclosure.

Below is a code sample of the login setup for testing in figure 113:

```
import { chromium } from '@playwright/test';
import dotenv from 'dotenv';

dotenv.config({ path: '.env.development' });

const TEST_USER_EMAIL = process.env.TEST_USER_EMAIL!;
const TEST_USER_PASSWORD = process.env.TEST_USER_PASSWORD!;

async function loginAndSaveState() {
  const browser = await chromium.launch({ headless: false }); // headless false = you see the browser
  const page = await browser.newPage();

  console.log('Navigating to Login page...');
  await page.goto('https://health-service.click/login');

  // Fill credentials
  await page.getByPlaceholder('name@example.com').fill(TEST_USER_EMAIL);
  await page.getByLabel('Password').fill(TEST_USER_PASSWORD);
  await page.getByRole('button', { name: /login/i }).click();

  // Wait for input-totp page
  await page.waitForURL(/\/input-totp/);

  console.log('Waiting for you to manually enter your TOTP...');
  console.log('You have 30 seconds to input TOTP manually.');
```

await page.waitForTimeout(30000); // Give you 30 seconds to enter OTP manually

```
  // Confirm you are on dashboard
  await page.waitForURL(/\/dashboard/);

  console.log('Logged in successfully! Saving storage...');

  // Save storage (cookies, localStorage) to a file
  await page.context().storageState({ path: 'tests/e2e/storageState.json' });

  console.log('Storage state saved at tests/e2e/storageState.json');

  await browser.close();
}

loginAndSaveState();
```

Figure 115 Login Testing Setup

### 6.3.2 Vitest Unit testing

Unit testing checks that functions, API calls, and React hooks work separately. Testing Library (2024) states that unit testing guarantees "small, focused parts of an application behave as expected and support building reliable, maintainable systems"<sup>1</sup>.

Vitest was the major frontend logic unit testing framework for PharmaLink. Vitest makes developing tests in Vite-powered apps fast, modern, and lightweight. Testing consisted of API call and React hook testing.

Authentication, patient, and prescription API functions were evaluated separately. API operations like `api.get`, `api.post`, `api.put`, and `api.delete` were mocked using `vi.spyOn` to simulate server responses without a live backend.

Key examples include:

- Authentication: Testing fetchUser, registerUser, login, verifyOTP, and logout API calls for valid payloads and answers.
- Patients API: Checking that getPatients, getPatientById, createPatient, deletePatient, and updatePatient returned proper patient data and interacted with API endpoints.
- Test Prescriptions API functions getPrescriptions, getPrescriptionById, createPrescription, and updatePrescriptionStatus for accurate behavior and request structure.

These tests verified that the frontend could interface with the backend and understand API responses.

Below is a code sample of a prescription unit test in figure114:

```
it('returns a prescription when given a valid ID', async () => {
  const mockData = { _id: '123', status: 'Pending' };

  vi.spyOn(api, 'get').mockResolvedValueOnce({ data: { data: mockData } });

  const result = await getPrescriptionById('123');

  expect(api.get).toHaveBeenCalledWith('/prescription/123');
  expect(result._id).toBe('123');
  expect(result.status).toBe('Pending');
});
```

Figure 116 Prescription Unit Test

React Query's QueryClientProvider and @testing-library/react's renderHook utility were used to unit test custom React hooks like useAuth, usePatients, usePatientById, and usePrescriptions.

Highlights are:

- Ensured useAuth provided user authentication and loading status.
- Verifying usePatients and usePrescriptions correctly retrieved patient and prescription lists, loaded states, and replied to backend data.
- Verifying mutation hooks like useUpdatePrescription and useUpdatePrescriptionStatus trigger API updates.
- More mocks were utilized to represent authenticated users and backend data without active external systems, speeding up and stabilizing tests.

Below is a code sample of a react hook unit test in figure 115:

```
it('calls updatePrescription with correct data', async () => {
  (
    updatePrescription as unknown as ReturnType<typeof vi.fn>
  ).mockResolvedValue({ updated: true });

  const { result } = renderHook(() => useUpdatePrescription('presc789'), {
    wrapper,
  });

  await act(async () => {
    await result.current.mutateAsync({ status: 'Updated' });
  });

  expect(updatePrescription).toHaveBeenCalled('presc789', {
    status: 'Updated',
  });
});
```

Figure 117 React Hook Unit Test

Below is a screenshot of all frontend unit tests passing in figure 116:

```
Liam@DESKTOP-D8RJA46 MINGW64 /d/IADTY4Work/Major-Project-Frontend (main)
$ npm run test

> react-cicd-app-test@0.0.0 test
> vitest

Generating routes...
✓ Processed routes in 156ms

RUN v3.0.4 D:/IADTY4Work/Major-Project-Frontend

stdout | tests/unit/auth/authentication.test.ts > Auth API calls > fetchUser returns user data
{ _id: 'user1', email: 'test@example.com' }

stdout | tests/unit/patients/patient.test.ts > Patient API calls > createPatient sends correct data and returns created patient
{ success: true, id: 'p789' }

✓ tests/unit/prescriptions/prescriptions.test.ts (5 tests) 9ms
✓ tests/unit/patients/patient.test.ts (5 tests) 10ms
✓ tests/unit/auth/authentication.test.ts (5 tests) 12ms
✓ tests/unit/auth/useAuth.test.tsx (1 test) 13ms
✓ tests/unit/prescriptions/usePrescriptions.test.tsx (6 tests) 98ms
✓ tests/unit/patients/usePatients.test.tsx (3 tests) 183ms

Test Files 6 passed (6)
Tests 25 passed (25)
Start at 17:16:14
Duration 3.30s (transform 305ms, setup 0ms, collect 3.11s, tests 324ms, environment 12.16s, prepare 892ms)
```

Figure 118 Unit Tests Passing

## 6.4 Backend Testing

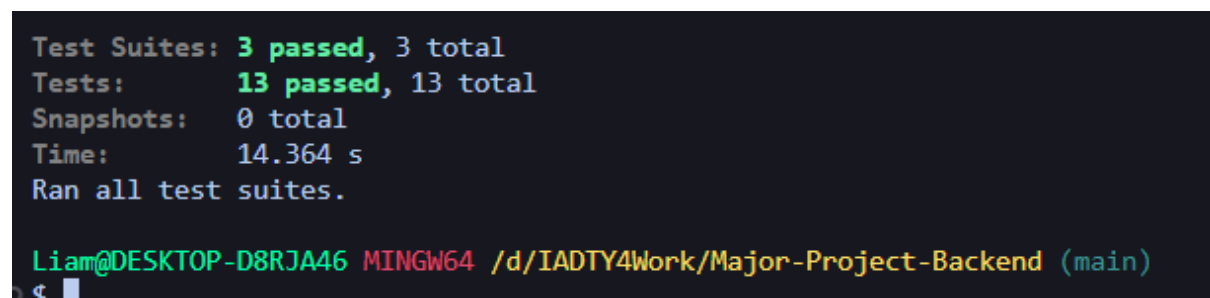
### 6.4.1 Unit Testing with Jest

PharmaLink tested backend logic, utilities, and API endpoints with Jest. Supertest was used to verify registration, login, and multi-factor authentication replies for `/api/auth/register`, `/api/auth/login`, and `/api/auth/login/mfa`. Tests confirmed that unauthenticated users could not access `/api/prescriptions`.

Utility functions were also thoroughly tested:

- **Hashing:** `hashField` and `compareField` correctly hashed and compared passwords.
- **JWT Creation:** `createJWT` generated valid JSON Web Tokens.
- **Encryption/Decryption:** `encryptData` and `decryptData` secured sensitive data and safely handled invalid inputs.

Below is a screenshot of the backend unit tests passing in figure 117:



```
Test Suites: 3 passed, 3 total
Tests:      13 passed, 13 total
Snapshots:  0 total
Time:       14.364 s
Ran all test suites.

Liam@DESKTOP-D8RJA46 MINGW64 /d/IADTY4Work/Major-Project-Backend (main)
$
```

Figure 119 Jest Unit Tests Passing



Below are code samples of testing JWTs and hashing utilities in figure 118:

```
describe('Hashing utilities', () => {
  it('should hash and verify a field correctly', async () => {
    const password = 'Password123!';
    const hashed = await hashField(password);
    const match = await compareField(password, hashed);

    expect(match).toBe(true);
  });

  it('should fail on wrong password', async () => {
    const password = 'Password123!';
    const hashed = await hashField(password);
    const match = await compareField('WrongPassword', hashed);

    expect(match).toBe(false);
  });
});

describe('JWT Creation', () => {
  it('should create a JWT token for a user', () => {
    const fakeUser = { id: '123', role: 'doctor' };
    const token = createJWT(fakeUser);

    expect(typeof token).toBe('string');
    expect(token.split('.').length).toBe(3); // JWT parts
  });
});
```

Figure 120 JWT & Hashing Tests

## 6.5 Conclusion

PharmaLink was reliable, secure, and tailored to real healthcare users thanks to testing.

Validated the system at every level using unit testing, integration testing, end-to-end testing, and user input. Vitest and Jest unit tests ensured that components, APIs, and business logic worked properly in isolation, whereas Playwright end-to-end testing verified user workflows from authentication to prescription administration.

Real-world users like pharmacists confirmed that PharmaLink solved Healthmail's problems. This user-centered evaluation showed that PharmaLink increased clinical workflow-aligned usability, security, and dependability.

The testing phase proved PharmaLink's technical stability and demonstrated its healthcare prescription management improvements. The system's quality, usability, and efficiency improved with thorough testing.

# 7 Project Management

## 7.1 Introduction

For this software Project to be planned, developed, and delivered successfully, effective project management was essential. This chapter describes the projects lifecycle in all its major stages, from the first proposal and requirement collection to design, implementation, and testing. It also emphasises the technologies that facilitate task tracking, version control, and communication during the development process.

This part also examines the projects general management, including communication with the project supervisor, difficulties encountered, and abilities developed. To give an in-depth overview of the experience obtained from overseeing a real world software development project, both technical and professional development are covered.

## 7.2 Project Phases

### 7.2.1 Proposal

The projects proposal phase outlined the fundamental goals, technological scope, and rationale for developing a secure, modern prescriptions transmission system between doctors and pharmacists. The "Pharmacist–Doctor Secure Prescriptions System" the proposal acknowledged the significant disadvantages of the healthcare industry current email-based communication systems, including Health mail, including problems with administrative burden, phishing threats, delays, and mis delivery.

A platform centered around a REST API was proposed as the solution, allowing doctors to assign prescriptions straight to pharmacists via a secure, real time digital system. This system would solve important issues with scalability, security, and efficiency and provide advantages like:

- End to end encrypted prescription transmission and storage.
- Realtime notifications for pharmacists.
- Role based access control (RBAC).
- Multifactor Authentication (MFA).
- Automated workflows that reduce manual overhead and improve reliability.

The proposal outlined an extensive list of objectives for the frontend and backend. GitHub Actions and AWS were used to set up the infrastructure, ESLint, Prettier, and Commitlint were used to enforce code quality, while Snyk and SonarQube were used for security tooling.

Additionally, it highlighted the necessity of employing strong testing techniques with Jest, Playwright, and Vitest.

The project suggested a React based online application with a UI driven by ShadCN components, TypeScript, and modern state management technologies. A Figma prototype would be used to create the frontends user friendly user flows, responsive user interface, and strict access control based on user roles.

### 7.2.2 Requirements

The requirements phase was important for determining the projects limits and direction. I was able to determine the advantages and disadvantages of the current prescription management systems by examining PioneerRx and WellSky. This information helped to shape the creation of a more specialised and user friendly solution.

Creating user personas for doctors and pharmacists assisted in focusing the design process on actual user requirements, directing features like prescription processes, patient management, and authentication. Frontend behaviour and API structure were directly influenced by the use case diagrams, which highlighted key system interactions.

The system was made sure to be feature complete, secure, scalable, and performance driven by separating needs into user, functional, technical, and nonfunctional categories. Architectural choices were directly driven by requirements such as role based access control, multifactor authentication, encryption, and the use of prescription data.

To make sure the system could satisfy user and industry requirements, the technical feasibility study further evaluated the stack and tools selected, ranging from CI/CD and testing frameworks to MERN and AWS.

With everything considered, this phase created a clear roadmap that eliminated uncertainty and set the way for efficient, user entered development.

### 7.2.3 Design

The PharmaLink systems design phase had a vital part in translating user requirements into a scalable and useful architecture. concentrated on developing a solution that could satisfy the functional goals while maintaining high performance, security, and user experience throughout the entire stack after determining what was necessary.

The three tier system architecture, which allowed for a clear separation of concerns between the frontend, backend, and data layers, was a significant result of this phase. This framework imposed secure access and communication between levels and significantly increased

scalability and maintainability. Utilising AWS cloud infrastructure together with technologies like React, Express, and MongoDB provided an approach that was both production ready and futureproof.

Usability and accessibility became the basis for frontend design decisions. A modular reliable client experience was established with the aid of ShadCN UI, Tailwind CSS, and technologies like TanStack Router and TanStack Query. While Zod and React Hook Form provided solid form validation, Figma was critical in building wireframes and visual hierarchy.

The Model Router Controller (MRC) design pattern was used to ensure structure and clarity for the backend. This pattern allowed me to build a maintainable and testable API while keeping business logic cleanly separated. Additionally, JWT, encryption using Node's native crypto module, Helmet, and express validator were integrated to emphasise the secure architecture. During this phase, workflows and relationships were clearly modelled using a variety of UML diagrams, including flowcharts, sequence diagrams, and ERDs. These tools acted as documentation for upcoming development and assisted in evaluating system logic before implementation.

Overall, this phase laid the foundation for building a robust, secure, and user friendly application, aligning tightly with the technical goals and user needs.

## 7.2.4 Implementation

Implementation turned architectural and design plans into a working PharmaLink system. A safe, scalable, and user-centered prescription management solution was my goal after the requirements and design phases.

With Node.js, Express, MongoDB, and React, full-stack development proved efficient. Good database schema design created clean relationships between users, patients, and prescriptions, supporting business logic and role-based workflows.

AWS deployment and secure environment variable management kept the application production ready. The frontend's modular architecture utilizing React and TanStack Router enabled dynamic and intuitive navigation, while the backend's Express and Model-Router-Controller (MRC) paradigm created a clean, maintainable API.

Security was crucial during implementation. Multifactor authentication (MFA), HTTP-only cookie session management, and field-level data encryption secured sensitive medical data. To ensure security, frontend and backend role-based access control was consistent.

Notification handling improved real-time updates and improved pharmacist and doctor processes. Comprehensive Jest, Vitest, and Playwright testing verified API endpoints, frontend components, and user journeys.

The Implementation phase turned PharmaLink from a technical design into a safe, production-ready prescription management system with careful planning and implementation.

## 7.2.5 Testing

PharmaLink's functionality, performance, and security have to be tested to meet project requirements.

Unit testing, end-to-end testing, and real user testing covered frontend and backend systems.

Vitest executed rapid, isolated unit tests on frontend components, React hooks, and API utility functions. This detected logic problems early in development.

Jest and Supertest tested authentication routes, encryption utilities, and session management operations on the backend to ensure system security and stability.

Playwright end-to-end testing verified login flows, role-based navigation, and prescription management across user roles.

Real pharmacist user testing revealed workflow efficiency improvements, particularly in navigation and role-based permissions.

PharmaLink ensured system stability, user experience, and security before deployment by combining automated and manual testing.

## 7.3 Project Management Tools

### 7.3.1 GitHub

GitHub was important to the PharmaLink systems development lifecycle and project management. GitHub functioned as a platform for task organisation, workflow automation, and progress monitoring during the whole development process, in addition to its primary function as a version control system.

<span>main</span> <span>5 Branches</span> <span>0 Tags</span>			<input type="text" value="Go to file"/>	<input type="button" value="Add file"/>	<input type="button" value="Code"/>
<b>Liam-Ronan-dev</b> Merge branch 'develop' <span>✖</span>			47974b8 · 20 hours ago <span>82 Commits</span>		
	.github/workflows	ci: updated deploy script	2 weeks ago		
	.husky	chore: fixing commitlint	2 months ago		
	docs	feat: re-structured ERD and prescriptions, connected to real ...	3 days ago		
	src	Merge branch 'develop'	20 hours ago		
	tests	configured eslint, prettier & jest, run on CI	2 months ago		
	.env.example	feat: added new error handler, request logger, and correctly ...	3 weeks ago		
	.gitignore	feat: added new error handler, request logger, and correctly ...	3 weeks ago		
	.prettierrc	feat: added medications functionality allowing pharmacists t...	last month		
	README.md	chore: testing nginx & EC2	2 months ago		
	commitlint.config.js	chore: updated GH secrets to not overwrite env file on ubun...	2 months ago		
	eslint.config.js	configured eslint, prettier & jest, run on CI	2 months ago		
	jest.config.js	configured eslint, prettier & jest, run on CI	2 months ago		
	lint-staged.config.js	test	2 months ago		
	package-lock.json	feat: re-structured ERD and prescriptions, connected to real ...	3 days ago		
	package.json	feat: re-structured ERD and prescriptions, connected to real ...	3 days ago		
	sonar-project.properties	Update sonar-project.properties	2 months ago		

Figure 121 GitHub Repository

The Gitbased version control system from GitHub made it easy to track changes, and rollbacks. With a main branch for code that was ready for production and a develop branch for testing features prior to merging, a clear branching strategy was put into place. The creation of feature specific branches improved focus and decreased merge conflicts by isolating functionality. GitHub Actions, one of GitHubs powerful features, was used in this project. These automated processes handled the following tasks based on push events:

- Unit testing, linting, formatting, and commit message checks are all examples of continuous integration (CI).
- Security scanning: Using programs like Snyk and SonarCloud to automatically find vulnerabilities.
- Executing Playwright test suites to guarantee UI dependability is known as end to end (E2E) testing.
- Deployment Pipelines: Upon successful builds, the frontend is automatically deployed to S3/CloudFront and the backend is automatically deployed to EC2.

All workflows

Filter workflow runs

Showing runs from all workflows

Help us improve GitHub Actions
Tell us how to make GitHub Actions work better for you with three quick questions.

Give feedback

251 workflow runs	Event	Status	Branch	Actor
<div> Merge branch 'develop' </div> <div> Deploy Back-end to EC2 Instance #82: Commit 47974b8 pushed by Liam-Ronan-dev </div> <div>main</div> <div> 20 hours ago 29s </div> <div>...</div>				
<div> Merge branch 'develop' </div> <div> Run Jest Tests &amp; Format Code #83: Commit 47974b8 pushed by Liam-Ronan-dev </div> <div>main</div> <div> 20 hours ago 16s </div> <div>...</div>				
<div> Merge branch 'develop' </div> <div> Security Scan #79: Commit 47974b8 pushed by Liam-Ronan-dev </div> <div>main</div> <div> 20 hours ago 1m 40s </div> <div>...</div>				
<div> fix: comments </div> <div> Deploy Back-end to EC2 Instance #81: Commit 3f02a7f pushed by Liam-Ronan-dev </div> <div>develop</div> <div> 20 hours ago 30s </div> <div>...</div>				
<div> fix: comments </div> <div> Run Jest Tests &amp; Format Code #82: Commit 3f02a7f pushed by Liam-Ronan-dev </div> <div>develop</div> <div> 20 hours ago 25s </div> <div>...</div>				
<div> fix: comments </div> <div> Security Scan #78: Commit 3f02a7f pushed by Liam-Ronan-dev </div> <div>develop</div> <div> 20 hours ago 1m 43s </div> <div>...</div>				
<div> fix: removed comment </div> <div> Run Jest Tests &amp; Format Code #81: Commit cb9ccfa pushed by Liam-Ronan-dev </div> <div>main</div> <div> yesterday 22s </div> <div>...</div>				
<div> fix: removed comment </div> <div> Security Scan #77: Commit cb9ccfa pushed by Liam-Ronan-dev </div> <div>main</div> <div> yesterday 1m 34s </div> <div>...</div>				
<div> fix: removed comment </div> <div> Deploy Back-end to EC2 Instance #80: Commit cb9ccfa pushed by Liam-Ronan-dev </div> <div>main</div> <div> yesterday 31s </div> <div>...</div>				

Figure 122 GitHub Actions Workflows

Overall, GitHub served as a complete project management platform in addition to a code repository, enabling quality control, automation, and open project lifecycle documentation. It was essential to sustaining a workflow for professional development because of its connection with testing frameworks, security analysis tools, and CI/CD technologies.

### 7.3.2 Notion

For analysing the projects daily and weekly progress, Notion was a crucial tool. It was perfect for monitoring development work, setting priorities, and keeping focus at every project phase because of its adaptability and user-friendly design.

Used a weekly tracker to write out important goals and deliverables at the beginning of each week. Could better prioritise activities and manage my time from looking back at the previous weeks' successes and deliverables.

Despite being the only developer on this project, being able to have the accountability and structure of a larger team by using Notion. In a single workspace, it replicated agile development techniques like sprint planning, progress monitoring, and retrospectives.

## 21st Jan - 27th Jan

- ✓ Re-write Project proposal
- ✓ Start setting up front-end CI/CD Infrastructure
  - ✓ S3 Bucket for static react app
  - ✓ Cloudfront distribution interacts with the bucket
  - ✓ Lambda@edge functions for setting security headers
  - ✓ SSL Certified using AWS certificate manager
  - ✓ Domain name purchases on route 53
- ✓ Created a dev env using github actions and an S3 bucket
- ✓ Setup Backend API ec2 instance with SSL cert from ACM and domain name from Route 53

Figure 123 Notion Tasks

## 7.4 Reflection

### 7.4.1 Personal Overview

thinking back on this projects journey, I can state with confidence that it has been the most ambitious and fulfilling software development experience I've worked on. I faced challenges in every aspect of building PharmaLink, a full stack, security focused prescription management system, from architectural design and technical problem solving to time management and making decisions under pressure.

This project involved more than just developing code; it involved resolving practical issues in an area where usability, security, and reliability are essential. I had to develop a security first mentality while working with sensitive data, such as patient information and medications, and consider carefully how to appropriately and correctly apply features like encryption, multifactor authentication, and access restriction.

The importance of process and planning was one of the main lessons learned. I had to approach development methodically because I had to oversee cloud infrastructure, CI/CD pipelines, backend APIs, and a dynamic frontend. Utilising platforms such as GitHub Actions, Synk, AWS, SonarCloud etc taught me how to work in a more polished, production ready setting.

There were numerous frustrating times, such as bugs, failing pipelines, or deployment problems. However, each challenge turned into a chance to gain new knowledge, whether it was



how to set up a reverse proxy server, write more scalable and effective code, or gain a deeper understanding of Node.js internals.

### 7.4.2 Project Development

This project was developed in an adaptable and iterative manner. I was aware of the scope and complexity of what I wanted to create from the beginning, but as development went on, I realised how much organisation, preparation, and discipline are needed to create a secure full stack application.

Flexibility was essential, but I started with clearly defined phases: planning, infrastructure setup, backend first, then frontend. Features like security or testing occasionally took longer than anticipated, and other times, design or user interface elements had to adapt in response to backend modifications. I had to constantly make little decisions, such as whether to push forward to fulfil a deadline or restructure code for maintainability. I learned from this how crucial it is to strike a balance between engineering best practices and efficiency.

The way I included technologies like Socket.IO for real time communication, AWS services for deployment, and MFA using OTPs was a significant highlight of the development phase. These were not little features; they required testing, trial and error, and study, but in the end, they improved the systems realism and professionalism.

In terms of technical growth, I learned far more than just writing APIs or React components. I learned about encryption strategies, proper model relationships in MongoDB, advanced routing with TanStack, Data fetching, and deployment workflows. Most importantly, I learned to think end to end from architecture to user experience to long term maintenance.

### 7.4.3 Project Oversight and Supervisor Communication

PharmaLink succeeded because to good communication and oversight. Weekly one-on-one meetings with my supervisor, John, kept momentum and priorities clear throughout development.

Each meeting checked the project's direction to avoid feature creep and technical diversions. John's problem-solving and critical thinking skills inspired me to think strategically about doctors' and pharmacists' real-world use cases rather than just coding features. I was challenged to consider whether each system design decision made sense from a user-centered rather than technical or engineering standpoint throughout our discussions.

A shift in mindset greatly affected the project's outcome. I prioritized workflows and interactions that fit with clinical practice, such as streamlining prescription creation for doctors and restricting pharmacists to relevant prescriptions.

John also advised on database model structure and relationships. Early conversations revealed where the initial data models needed improved normalization and where key interactions (such as patients, prescriptions, and assigned pharmacists) needed strict enforcement to preserve security and consistency. He encouraged solid database architecture from the start to avoid problems in the future.

Consistent supervisor communication turned the project from a technical project to a meaningful, real-world solution. The weekly 1:1 sessions meant that crucial decisions were made intentionally with user needs in mind, improving system functionality and credibility.

#### 7.4.4 Technical Skills

I learned a lot about frontend, backend, cloud infrastructure, and modern development practices while developing PharmaLink.

On the frontend, I learned TanStack Router and TanStack Query for dynamic client-side routing and advanced server-state caching. I also got stronger at form validation, API integration, and developing clean, reusable components that worked well with the backend in React and TypeScript.

Clean code was prioritized during implementation. Wrote improved modular, well-organized code, followed single-responsibility principles, and made the project easier to scale and maintain.

Acquired knowledge of EC2, S3, Route 53, and CloudFront to manage deployment, storage, DNS, and global content delivery on the cloud and backend. Creating an AWS environment helped me understand scalable infrastructure and production deployment procedures.

Growth also took place during testing. Vitest, Playwright, Jest, and Supertest showed me how to build efficient unit tests, broad user flow tests, and backend API tests. Adding these testing frameworks to the development cycle helped me create more reliable and production-ready software.

Security principles were crucial to the project. MFA, RBAC, data encryption, and cookie-based web application session security were my practical experiences. Investigated web sockets for real-time features to improve pharmacist and doctor system interaction.

In development, set up CI/CD pipelines with GitHub Actions and used Husky, ESLint, Prettier, and lint-staged to ensure consistent, high-quality code before merging changes.

Also, learned to parse XML into JSON for prescription data imports, improving my data transformation and backend API integration skills.

This project helped me learn modern technologies, scalable system architecture, secure authentication, and cloud deployment best practices, accelerating my full-stack development experience.

### 7.4.5 Further Competencies and Professional Skills

PharmaLink taught me several professional skills beyond technical coding. Planning development phases, prioritizing security and access control, and maintaining weekly progress increased project management abilities. Splitting the project into milestones and reviewing priorities during supervisor meetings kept it focused and prevented scope creep.

Regular reviews of system design decisions, including database modelling, role-based access control, and real-world user workflows, improved critical thinking and problem-solving.

Enhanced the project's usability and relevancy by thinking strategically about how doctors and pharmacists would use the system.

Multifactor authentication, cookie-based session management, and encrypted data processing raised security awareness. Knowing that security is a need for every component of the system changed how I approached backend, frontend, and deployment responsibilities.

Finally, PharmaLink deployed to AWS, setting up EC2, S3, Route 53, and CloudFront, and developing a CI/CD pipeline with GitHub Actions improved my cloud infrastructure and DevOps skills. Integrating code quality tools like Husky, ESLint, Prettier, and lint-staged helped maintain professional development standards throughout the project.

## 7.5 Conclusion

PharmaLink's success depended on project management. During requirements and design, careful planning created a path for development that met user needs and industry standards. Supervisor meetings kept the project on track and encouraged realistic, user-centered decision making throughout.

Continuous communication, planned milestones, and critical technical assessment lowered risks, managed complexity, and maintained momentum. This approach enables the project to adapt to new technical obstacles and possibilities without losing sight of key aims.

PharmaLink was feature complete, resilient, secure, and scalable due to project management practices.

## 8 Conclusion

The PharmaLink project was difficult, gratifying, and transforming technically and professionally. From the start of requirements gathering and planning, an organized, careful approach was required to deliver a system that truly met doctors' and pharmacists' needs. Deeply understanding user stories, workflows, and security needs shaped every design and development decision.

The design process was essential also. Early design of system architecture, data models, and user experience flows streamlined development and prevented severe architectural challenges. Careful upfront design allowed role-based access control, multifactor authentication, and real-time notifications to be implemented cleanly and uniformly throughout the program.

The plan emphasised problem-solving. Practical, critical thinking was needed to improve database relationships, enforce security at every level, and modify frontend logic to fit real-world user behaviors. Weekly communication with my supervisor, John Montayne, was key to success. His strategic guidance and encouragement for including technical and real-world considerations improved my decision-making and kept the project practical.

My technical skills improved across the stack. On the backend, I gained knowledge to design secure APIs with Node.js and Express, manage sessions with secure cookies, and set up strong authentication and authorization.

My React, TypeScript, TanStack Router, and TanStack Query skills improved, allowing me to develop a modular, scalable, and dynamic user interface.

Deploying the system using AWS services like EC2, S3, Route 53, and CloudFront gave me hands-on experience with cloud infrastructure, deployment pipelines, and production environments, which will be significant in my career.

In the future, I would like to add a doctor-pharmacy query system. Pharmacists might easily request prescription clarifications using the application, increasing communication, delays, and patient safety.

Overall, I am satisfied with the final PharmaLink application. It delivers the key functionality envisioned at the beginning of the project while prioritising security, usability, and scalability.

This project improved my coding, architectural thinking, security awareness, cloud deployment, and professional communication, along with technical achievements.

## References

- Assal, H., & Chiasson, S. (2018). Open access to the Proceedings of the Fourteenth Symposium on Usable Privacy and Security is sponsored by USENIX. Security in the Software Development Lifecycle Security in the Software Development Lifecycle.
- <https://www.usenix.org/system/files/conference/soups2018/soups2018assal.pdf>
- Balancing Code Quality and Security: A Practical Guide. (2024, August). Java Tech Blog.
- <https://javanexus.com/blog/balancingcodequalitysecurityguide>
- Franke, L., Liang, H., Farsanehpour, S., Brantly, A., Davis, J. C., & Brown, C. (2024). An Exploratory MixedMethods Study on General Data Protection Regulation (GDPR) Compliance in OpenSource Software. ArXiv.org. <https://arxiv.org/abs/2406.14724>
- Kantarcioglu, M., & Ferrari, E. (2019). Research Challenges at the Intersection of Big Data, Security and Privacy. *Frontiers in Big Data*, 2.
- <https://doi.org/10.3389/fdata.2019.00001>
- Khan, R. A., Khan, S. U., Khan, H. U., & Ilyas, M. (2022). Systematic Literature Review on Security Risks and its Practices in Secure Software Development. *IEEE Access*, 10, 5456–5481. <https://doi.org/10.1109/ACCESS.2022.3140181>
- Md Abul Khair. SecurityCentric Software Development: Integrating Secure Coding Practices into the Software Development Lifecycle. *Technology & Management Review*, 2018, 3 (1), pp.1226. hal04565385
- Koo, J., Kang, G., & Kim, Y.G. (2020). Security and Privacy in Big Data Life Cycle: A Survey and Open Challenges. *Sustainability*, 12(24), 10571. MDPI.
- <https://doi.org/10.3390/su122410571>
- Mastering Code Quality and Application Security: A Comprehensive Guide for Developers to Secure Coding Practices. (2024). Aptori.dev; Aptori.
- <https://aptori.dev/guide/masteringcodequalityandapplicationsecurity>
- Moscher, M. (2017). Continuous Compliance Testing. (Masters thesis). RWTH Aachen University, Aachen, Germany. [https://swc.rwthachen.de/theses/continuouscompliance/2017\\_Moscher\\_ContinuousComplianceTesting\\_\\_FINAL.pdf](https://swc.rwthachen.de/theses/continuouscompliance/2017_Moscher_ContinuousComplianceTesting__FINAL.pdf)
- NegriRibalta, C., Marius LombardPlatet, & Salinesi, C. (2024). Understanding the GDPR from a requirements engineering perspective a systematic mapping study on regulatory data protection requirements. *Requirements Engineering*.
- <https://doi.org/10.1007/s00766024004234>

- Potter, B., & McGraw, G. (2004). Software Security Testing. IEEE Security & Privacy, 2(5), 81–85. <https://doi.org/10.1109/MSP.2004.84>
- Securing CI/CD Pipeline: Automating the Detection of Misconfigurations and Integrating Security Tools. (n.d.). Retrieved January 9, 2024, from <https://norma.ncirl.ie/6529/1/muskanmangla.pdf>
- Theurich, P., Witt, J., & Richter, S. (2023). Practices and Challenges of Threat Modelling in Agile Environments. Informatik Spektrum, 46(4), 220–229. <https://doi.org/10.1007/s00287023015495>
- ValdésRodríguez, Y., HochstetterDies, J., DíasArancibia, J., & CadenaMartínes, R. (2023). Towards the Integration of Security Practices in Agile Software Development: A Systematic Mapping Review. Applied Sciences, 13(7), 4578. <https://doi.org/10.3390/app13074578>
- View of SecurityFirst Approaches to CI/CD in CloudComputing Platforms: Enhancing DevSecOps Practices. (2024). Sydneyacademics.com. <https://sydneyacademics.com/index.php/ajmlra/article/view/131/126>
- GeeksforGeeks. (2023, November 24). *Use Case Diagram*. GeeksforGeeks. <https://www.geeksforgeeks.org/usecasediagram/>
- Altexsoft. (2023, November 30). *Functional and Nonfunctional Requirements: Specification an*. AltexSoft. <https://www.altexsoft.com/blog/functionalandnonfunctionalrequirementsspecificationandtypes/>
- Gupta, R. (2024, February 23). *Software Architecture Patterns: What Are the Types and Which Is the Best One for Your Project*. Www.turing.com. <https://www.turing.com/blog/softwarearchitecturepatternstypes>
- Sommerville, I. (2015). *Software Engineering* (10th ed.). Pearson. Discusses the role of requirements engineering in successful software development and project management.
- Pohl, K. (2010). *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer. Provides an indepth understanding of the requirements gathering process, including elicitation, validation, and management.
- Wiegers, K. E., & Beatty, J. (2013). *Software Requirements* (3rd ed.). Microsoft Press. Covers best practices for gathering and managing software requirements in different project environments.

- Chu, M. (2023, December 1). *Why Personas are Important in Software Development*. Techblocks.  
<https://tblocks.com/articles/whypersonasareimportantinsoftwaredevelopment/>
- geeksforgeeks. (2017). *Software Design Patterns* GeeksforGeeks. GeeksforGeeks.  
<https://www.geeksforgeeks.org/software-design-patterns/>
- Amazon Web Services. (n.d.). *What is RESTful API? RESTful API Beginners Guide* AWS. Amazon Web Services, Inc. <https://aws.amazon.com/what-is/restful-api/>
- GeeksforGeeks. (2017, October 27). *Unified Modeling Language (UML) | Sequence Diagrams*. GeeksforGeeks.  
<https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/>
- Flowcharts in Programming Applications & Best Practices. (n.d.).  
Www.senflowchart.com.  
<https://www.senflowchart.com/guides/flowcharts-in-programming>
- <https://medium.com/@amanuelabraham0202/flowcharts-for-programmers-2aff6a6d8f63>
- LucidChart. (2024). *What is an Entity Relationship Diagram (ERD)?* Lucidchart.  
<https://www.lucidchart.com/pages/er-diagrams>
- Amazon Web Services. (2023a). *Lambda@Edge – Run your code closer to your users*. Retrieved from <https://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html>
- Amazon Web Services. (2023b). *Using Lambda@Edge to add HTTP security headers*. Retrieved from:  
<https://aws.amazon.com/blogs/networking-and-content-delivery/adding-http-security-headers-using-lambda-edge-and-cloudfront/>
- MDN Web Docs. (2023). *Content Security Policy (CSP)*. Retrieved from <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>
- Mozilla Developer Network. (2023). *XFrameOptions*. Retrieved from <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/XFrameOptions>
- OWASP. (2021). *HTTP Headers – A guide to securing your web application*. Retrieved from <https://owasp.org/www-project-secure-headers/>
- Sommerville, I. (2016). *Software engineering* (10th ed.). Harlow, England: Pearson Education.
- GeeksforGeeks. (2023). *UML Sequence Diagram*. Retrieved from <https://www.geeksforgeeks.org/uml-sequence-diagram/>
- Lucidchart. (2023). *What is a flowchart? Process flow diagram explained*. Retrieved from <https://www.lucidchart.com/pages/what-is-a-flowchart>



- Amazon Web Services. (2023). *What is Amazon EC2?* Retrieved from <https://aws.amazon.com/ec2/>
- Amazon Web Services. (2023). *Elastic Load Balancing features*. Retrieved from <https://aws.amazon.com/elasticloadbalancing/features/>
- Amazon Web Services. (2023). *Amazon Route 53 – Scalable Domain Name System*. Retrieved from <https://aws.amazon.com/route53/>
- Amazon Web Services. (2023). *AWS Certificate Manager*. Retrieved from <https://aws.amazon.com/certificatemanager/>
- Amazon Web Services. (2023). *Amazon S3: Object storage built to retrieve any amount of data*. Retrieved from <https://aws.amazon.com/s3/>
- Amazon Web Services. (2023). *What is Amazon CloudFront?* Retrieved from <https://aws.amazon.com/cloudfront/>
- Amazon Web Services. (2023). *AWS WAF – Web Application Firewall*. Retrieved from <https://aws.amazon.com/waf/>
- Amazon Web Services. (2023). *Using Lambda@Edge to add HTTP security headers*. Retrieved from <https://aws.amazon.com/blogs/networkingandcontentdelivery/addinghttpsecurityheadersusinglambdaedgeandcloudfront/>
- Fowler, M. (2006). *Continuous Integration*. Retrieved from <https://martinfowler.com/articles/continuousIntegration.html>
- GitHub Docs. (2024). *Understanding GitHub Actions*. Retrieved from <https://docs.github.com/en/actions/learngithubactions/understandinggithubactions>
- Snyk. (2023). *Find and fix vulnerabilities in open source dependencies*. Retrieved from <https://snyk.io/>
- SonarSource. (2023). *SonarCloud documentation*. Retrieved from <https://docs.sonarcloud.io/>
- GitHub Docs. (2024). *GitHub Actions – Security automation*. Retrieved from <https://docs.github.com/en/actions/securityguides/securityhardeningforgithubactions>
- ESLint. (2024). *Find and fix problems in your JavaScript code*. Retrieved from <https://eslint.org/>
- Playwright. (2024). *Playwright for endtoend testing*. Retrieved from <https://playwright.dev/>
- Microsoft. (2024). *Playwright Testing Framework Overview*. Retrieved from <https://playwright.dev/>

- Vitest. (2024). *A blazing fast unit test framework powered by Vite*. Retrieved from <https://vitest.dev/>
- Vite. (2024). *Vite – Next Generation Frontend Tooling*. Retrieved from <https://vitejs.dev/>
- Human Medicines Authorised Products Latest list of Authorised or Transfer Pending Products data.gov.ie. (2018). Data.gov.ie. <https://data.gov.ie/dataset/medicinesauthorisedortransferpendingproducts/resource/6987c2af0c4048da820776b5da141266>
- National Institute of Standards and Technology. (2001). *Announcing the Advanced Encryption Standard (AES) (FIPS PUB 197)*. U.S. Department of Commerce. Retrieved from <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf>
- Socket.io. (n.d.). *Socket.IO Documentation*. Retrieved from <https://socket.io/docs>
- Nodemailer. (n.d.). *Nodemailer – Easy as cake email sending from Node.js*. Retrieved from <https://nodemailer.com>
- Health Products Regulatory Authority. (2024). *Medicines: authorised or transfer pending products [XML dataset]*. Retrieved from <https://data.gov.ie/dataset/medicinesauthorisedortransferpendingproducts>
- Harley, A. (2015, February 16). *Personas Make Users Memorable for Product Team Members*. Nielsen Norman Group. <https://www.nngroup.com/articles/persona/>
- BrowserStack. (2023). *Frontend Testing: What is it & How to Perform it?* Retrieved from <https://www.browserstack.com/guide/frontend-testing>
- Microsoft Learn. (2023). *End-to-end testing overview*. <https://learn.microsoft.com/en-us/azure/devops/pipelines/test/end-to-end-testing>
- Testing Library. (2024). *What is unit testing?* Retrieved from <https://testing-library.com/docs/guiding-principles/>