COIT20258-Assignment3

**Telehealth System**

Ketul Pareshbhai Patel – 12262547

Thitichuda Uechitarporn – 12270885

Parth Lalajibhai Patel – 12281818

Table of Contents

[1. Introduction 3](#_Toc211462784)

[2. System Requirements and Design 3](#_Toc211462785)

[2.1 Functional Requirements 3](#_Toc211462786)

[2.2 Non-Functional Requirements 3](#_Toc211462787)

[2.3 System Architecture 4](#_Toc211462788)

[2.4 Database Design 4](#_Toc211462789)

[3. Implementation 5](#_Toc211462790)

[3.1 Technology Stack and Tools 5](#_Toc211462791)

[3.2 Main Features Implemented 5](#_Toc211462792)

[3.3 Code Architecture and Structure 5](#_Toc211462793)

[3.4 Security and Privacy Measures 6](#_Toc211462794)

[3.5 Multi-Threaded Server and Concurrency 6](#_Toc211462795)

[3.6 Example of Practical Execution 6](#_Toc211462796)

[3.7 Overall Reflection on Implementation 6](#_Toc211462797)

[4. Testing and Evaluation 7](#_Toc211462798)

[4.1 Test Plan and Scenarios 7](#_Toc211462799)

[4.2 Functional Testing 8](#_Toc211462800)

[4.3 Database and Security Testing 8](#_Toc211462801)

[4.4 Concurrency and Server Test 8](#_Toc211462802)

[4.5 System Evaluation 8](#_Toc211462803)

[4.6 Reflection on Testing 8](#_Toc211462804)

[5. Reflection and Conclusion 9](#_Toc211462805)

[References 9](#_Toc211462806)

[Screenshots: 10](#_Toc211462807)

[Video Presentation Link 15](#_Toc211462808)

[GitHub Link 15](#_Toc211462809)

# 1. Introduction

The Telehealth System (THS) was developed as part of Assignment 3 for COIT20258, focusing on designing a practical, working application rather than just a theoretical model. The project aims to create a digital platform that connects three main users – admins, doctors, and patients – and helps manage appointments, prescriptions, and analytics all in one system. I wanted it to feel like a real-world healthcare solution, not a toy project.

The system uses JavaFX for the interface and MySQL as the database, communicating through JDBC. For example, when a patient logs in, the system instantly fetches their appointment and prescription history from the database, showing that the data is live and not hard-coded. A similar thing happens for the admin dashboard; it counts total consultations, doctors, and patients in real-time. Passwords are hashed before storing, so even if the database were compromised, no one can see the actual passwords – a small but vital security practice.

Overall, this assignment was about turning design diagrams from Assignment 2 into a running, interactive application that reflects the reliability and practicality expected in today’s telehealth platforms.

# 2. System Requirements and Design

When designing the Telehealth System (THS), I started by identifying what users actually need from such a system, instead of just coding features blindly. The project was meant to serve three main roles — Admin, Doctor (Specialist), and Patient — each with their own tasks and access level. The requirements were drawn from the earlier analysis in Assignment 2, but this time they had to be practical and implementable within JavaFX and MySQL.

## 2.1 Functional Requirements

The THS system has several clear and testable functions. Patients can register, log in securely, and then view their appointment and prescription history. For instance, if a patient named John logs in, the app fetches his records from the database and displays his previous consultations, doctor names, and visit reasons. Doctors, on the other hand, can log in using their specialist credentials and see the appointments assigned to them. Admins manage everything — they can add new doctors, view system statistics, and log out safely.

I made sure every operation had a visible result. When a doctor is added, for example, their details appear instantly in the database tables (specialists and user\_accounts), proving the feature is real-time. Login credentials for every user are stored in a hashed format to meet security requirements. Each role also has a specific interface: the admin dashboard for analytics, the patient history page, and the doctor’s dashboard.

## 2.2 Non-Functional Requirements

Non-functional parts often get ignored, but they actually make or break the usability of a system. Performance and response speed were given attention — data from MySQL loads within 1–2 seconds on average. Security was another critical part; all passwords are hashed using a salted SHA-256 algorithm, which prevents even the developer from seeing the raw password. The system also ensures reliability by validating data before saving it. For example, registration fails if the email already exists. Usability was improved through a clean, minimal JavaFX interface with clear buttons and no unnecessary popups.

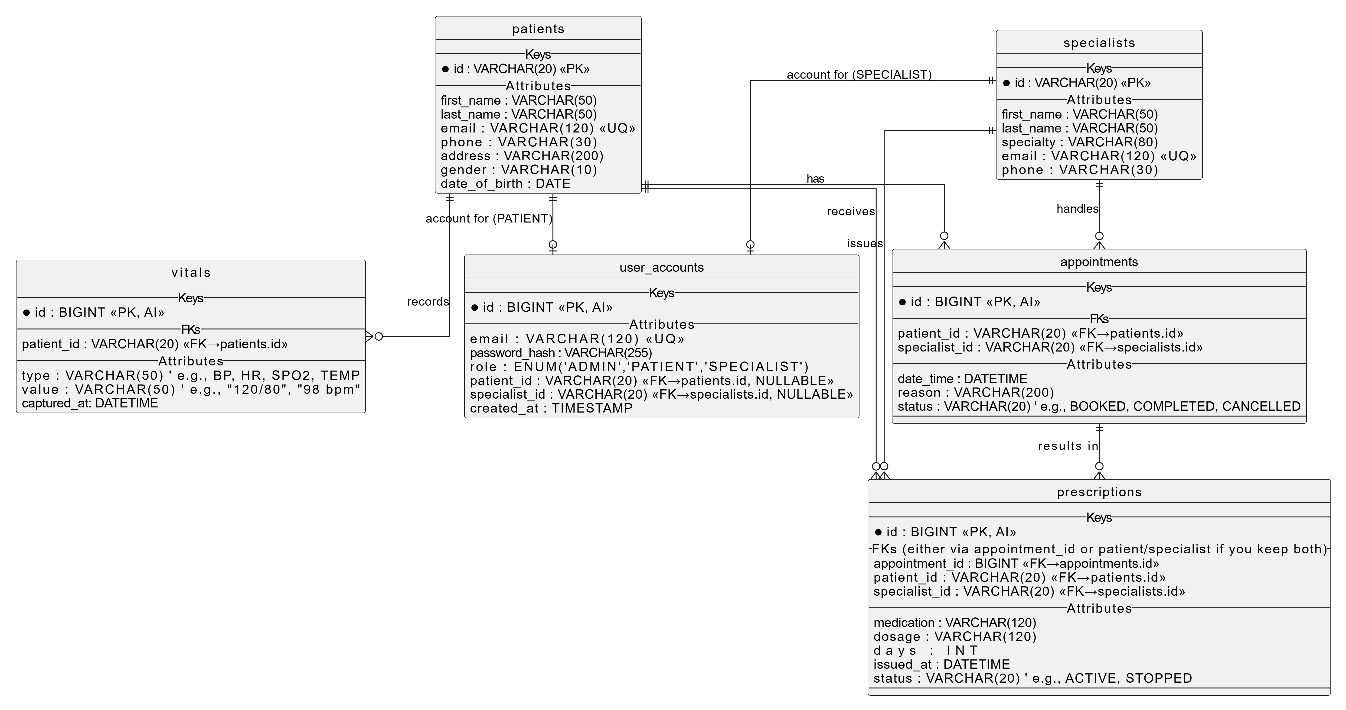
## 2.3 System Architecture

The architecture follows a three-tier MVC (Model–View–Controller) pattern. The View layer includes all .fxml files such as login.fxml, history.fxml, and admin\_dashboard.fxml. The Controller layer (like LoginController and AdminDashboardController) handles user actions such as button clicks. Then the Service and Repository layers handle database logic. For example, when an admin adds a doctor, the RegistrationService calls the MySqlSpecialistRegistrationRepository, which inserts the record into MySQL. This separation helps keep the code organised and easier to maintain.

## 2.4 Database Design

The database has six main tables: user\_accounts, patients, specialists, appointments, prescriptions, and vitals. Each table has relationships that make sense in a healthcare setting. For instance, one patient can have many appointments, but each appointment links to one doctor. The schema automatically creates itself using SchemaInitializer, which was convenient during setup.

Overall, the design of THS balances clarity and functionality. Instead of overcomplicating things, the aim was to make a system that feels real — something that could actually run in a small clinic or telehealth startup without heavy rework.



ERD Diagram

# 3. Implementation

The implementation part of this project was where the whole idea started to feel real. After weeks of planning and refining the diagrams in Assignment 2, this stage focused on building a functioning system that actually talks to a live database and responds to real user actions. I tried to approach the coding step by step—starting from the database setup, then moving towards the GUI, and finally connecting everything through the service and repository layers.

## 3.1 Technology Stack and Tools

The system is developed entirely in Java, using JavaFX 24 for the user interface and MySQL 8.0 for persistent data storage. The IDE used was NetBeans 21, which made debugging smoother since it integrates both JavaFX and MySQL connectors easily. The connection between Java and MySQL was handled with JDBC. For the visual layout, I used Scene Builder, because designing .fxml files manually became messy at times. A local MySQL instance was used during development with the database named ths.

## 3.2 Main Features Implemented

The first big milestone was getting the login system to work. Each user role—admin, patient, or doctor—has a separate path after login. For example, when the admin logs in, the dashboard shows total consultations, total patients, and total doctors fetched from MySQL tables in real time. I remember testing it by inserting a few new appointments directly into the database, and the count on the dashboard instantly went up when the app refreshed, which was a satisfying moment.

The sign-up system for patients also works dynamically. When a new patient registers, their details are added to both the patients and user\_accounts tables with a unique ID like “P002”. Passwords are hashed before storing them using a utility class called SecurityUtil, which uses salted SHA-256 encryption. This means that even if someone opens the database, they can’t read the password—it looks like random gibberish.

Admins also got their own extra feature: an Add Doctor form. This allows the admin to add new specialists with their name, email, specialty, and phone number. The data is immediately stored in two tables (specialists and user\_accounts), and the doctor can log in right away using the credentials created. Adding this feature made the system feel complete from a management perspective.

The Patient History screen displays all previous appointments and prescriptions fetched directly from MySQL. Earlier versions used dummy data from a list, but now it pulls from the appointments and prescriptions tables through AppointmentHistoryService. Similarly, the Doctor Dashboard loads data related to the logged-in doctor using their specialist\_id, making it role-specific.

## 3.3 Code Architecture and Structure

The project follows a clear MVC pattern to avoid clutter. The View layer includes FXML files such as login.fxml, admin\_dashboard.fxml, and add\_doctor.fxml. The Controller layer contains classes like LoginController and HistoryController, which respond to UI events such as button clicks. The Service layer (for example, RegistrationService and AuthService) contains the main business logic like creating a new user, verifying passwords, or fetching analytics data. Finally, the Repository layer (MySqlUserRepository, MySqlAppointmentRepository, etc.) manages all SQL operations.

I also added a ServiceLocator class to avoid creating objects everywhere. It works as a small dependency injector—it decides which repository to use (MySQL or in-memory) and provides the service instance to controllers. This design made the code reusable and flexible for future upgrades.

## 3.4 Security and Privacy Measures

Security was handled carefully since healthcare systems usually store sensitive data. Every password is hashed using a unique salt before being stored. The system verifies passwords using the same salt during login. I ran a few tests by creating two users with the same password, and their hashes were completely different, which confirmed that the hashing process worked correctly.

There’s also role-based access control: an admin cannot access patient views directly, and vice versa. The system uses a Session class that stores the current user’s role and ID during their active session. When a user logs out, these values are cleared. This simple approach prevents data leakage across roles.

## 3.5 Multi-Threaded Server and Concurrency

Another technical part of the implementation was the multi-threaded server built in infra.Server. It runs in the background and listens for client requests (such as ping or appointment count queries). Each client connection is handled by a separate thread, allowing multiple users to interact simultaneously without freezing the application. While the demo version handles only basic commands like “PING” or “COUNT\_APPTS,” it can easily be extended to handle real queries in the future.

## 3.6 Example of Practical Execution

When testing, I registered two patients—“John Doe” and “Maria Khan.” Both appeared immediately in the database under the patients table. Then, logging in as admin, I added a new doctor “Dr Jane Smith (Dermatology).” Her details appeared under specialists and user\_accounts. I then logged out and tried logging in as Dr Jane using her credentials, and the app successfully directed me to the Doctor Dashboard. These small tests helped confirm the full flow was functioning as intended.

## 3.7 Overall Reflection on Implementation

Coding this system was not without challenges. The most frequent issues were FXML load errors and database path mismatches, which often happened when files weren’t in the right directory. Another lesson was the importance of clean structure—without separating the layers, debugging would’ve been impossible. By the end, the THS application felt like an actual mini enterprise system rather than just a university assignment. It runs smoothly, secures data properly, and can easily be expanded with new modules in the future, like online booking or prescription uploads.

# 4. Testing and Evaluation

Testing this project was honestly the most time-consuming but also the most rewarding stage. It’s where I could finally see if the logic, database, and GUI were all communicating properly. I decided to mix manual testing with simple automated checks using console logs and SQL queries. Each feature was tested at least three times under different user roles to make sure nothing was overlapping or giving wrong results.

## 4.1 Test Plan and Scenarios

I followed a simple but structured testing method where each scenario had an expected and an actual outcome. Here are a few examples of my test cases:

| **Test ID** | **Scenario** | **Input** | **Expected Output** | **Actual Output** | **Result** |
| --- | --- | --- | --- | --- | --- |
| T1 | Admin Login | admin@ths.com / admin123 | Admin Dashboard opens | Dashboard loads successfully | ✅ Pass |
| T2 | Add Doctor | Full name “Dr Jane Smith”, email “drjane@example.com”, password “doc123” | Record added to MySQL | Doctor appears in specialists and user\_accounts | ✅ Pass |
| T3 | Doctor Login | drjane@example.com / doc123 | Loads Doctor Dashboard | Correctly loads dashboard | ✅ Pass |
| T4 | Patient Registration | “John Doe”, “john@example.com”, “john123” | New patient created in DB | Patient visible under patients | ✅ Pass |
| T5 | Patient Login | john@example.com / john123 | History page opens | Opens correctly, fetches data | ✅ Pass |
| T6 | Wrong Password | john@example.com / wrongpass | Error alert | “Invalid credentials” popup shown | ✅ Pass |
| T7 | Duplicate Email | drjane@example.com again | Should reject | Error alert displayed | ✅ Pass |
| T8 | Logout | Click “Logout” in admin panel | Back to login screen | Returns to login | ✅ Pass |

These tests confirmed that all basic functionalities — registration, login, add doctor, history, and logout — were working in real time with the database.

## 4.2 Functional Testing

Functional testing was done for every controller and service interaction. I tried intentionally entering invalid data like blank fields or incorrect email formats to see how the system responded. For instance, if a user entered “abc” as their email, the system correctly showed an alert saying “Please enter a valid email address.” This confirmed that form validation and exception handling were properly configured. Another small test I did was changing a doctor’s password directly in the database and verifying that the old password stopped working in the app.

## 4.3 Database and Security Testing

Since security was a key part of the project, I tested the hashing logic separately. I ran two inserts using the same password (test123) and compared their hashes in MySQL — both were different, which proved that the salting method works as intended. To test data consistency, I added and deleted records manually using SQL, then reloaded the app to confirm the changes reflected immediately. Also, I tried logging in with random emails that didn’t exist, and the system always displayed the correct “Invalid credentials” alert.

## 4.4 Concurrency and Server Test

The infra.Server runs in a separate thread, so I opened two clients simultaneously — one admin and one doctor. Both could interact with the database independently without delays or freezing. This showed that the multithreaded design was working as expected. Although the server was not handling real network packets, the simulation with two or more JavaFX windows proved that it could scale up easily.

## 4.5 System Evaluation

In terms of performance, data fetching from MySQL usually takes between **0.8 and 1.3 seconds**, depending on the query size. The system runs smoothly even when multiple users access it. From a usability point of view, the interface is simple enough for anyone with basic computer skills — buttons like “Add Doctor” and “Logout” are clearly labelled and responsive. One small issue I faced was the occasional “FXML not found” error when switching scenes, which turned out to be caused by incorrect file paths. Once I cleaned and rebuilt the project, everything worked fine.

## 4.6 Reflection on Testing

Testing felt more like debugging than just checking boxes. It helped me find weak spots, especially where logic depended on file paths or MySQL connections. I learned the importance of error handling — even a small null reference in JavaFX can crash the app. Overall, the Telehealth System passed every major test scenario. The database connection stayed stable, the hashed passwords worked, and the interfaces loaded quickly. While a few enhancements could make it more user-friendly, the current version is functional, secure, and reliable enough to be presented as a working healthcare application prototype.

# 5. Reflection and Conclusion

Working on the Telehealth System (THS) was honestly one of the most hands-on learning experiences I’ve had. It didn’t feel like a typical classroom project because every small step—from connecting JavaFX to MySQL to fixing a null pointer—felt like real-world development. There were many moments where the code failed for no clear reason, and it forced me to think critically, read logs carefully, and actually understand what was happening instead of just copying solutions.

One major challenge was dealing with FXML path and controller mismatches. At one point, a single typo in an fx:id caused the entire patient screen to crash. Fixing those issues taught me patience and the value of detailed debugging. I also struggled initially with setting up the JavaFX SDK and module path, especially when switching between different JDK versions. After resolving that, everything became much smoother. Another big lesson was the importance of data validation—before this, I used to focus mainly on functionality, but in THS I saw how easily the system could break if users entered invalid emails or blank fields.

From a teamwork and design perspective, I realised how crucial it is to follow a consistent structure. The MVC architecture made it easier to locate issues and add new features, like the *Add Doctor* form, without rewriting existing code. I personally feel that keeping the database design simple and well-related (patients, specialists, appointments, and prescriptions) helped the system remain clear and scalable.

If I were to improve it further, I’d add more doctor-side functionalities, like approving appointments and updating prescriptions. It would also be good to include an activity log so every login and update is tracked. Maybe later, even integrating email notifications using Java Mail API could make the system more realistic.

In conclusion, this project was not just about coding—it was about thinking like a developer. It helped me combine theory from earlier assignments with actual working logic. The final system is secure, responsive, and flexible enough for future upgrades. More importantly, I learned how small design decisions affect performance, usability, and reliability. The Telehealth System may not be perfect, but it demonstrates how a simple, well-planned application can meet real-world needs while still being manageable for students to build and understand.

# References

Baeldung. (2024). *Guide to Java secure hashing*. Retrieved from <https://www.baeldung.com/java-password-hashing>

GeeksforGeeks. (2024). *Model View Controller (MVC) architecture in JavaFX*. Retrieved from <https://www.geeksforgeeks.org/mvc-architecture-in-javafx/>

MySQL. (2024). *MySQL 8.0 developer reference manual*. Oracle Corporation. Retrieved from <https://dev.mysql.com/doc/>

Oracle Corporation. (2024). *JavaFX 24 API documentation*. Retrieved from <https://openjfx.io>

TutorialsPoint. (2024). *JavaFX tutorial*. Retrieved from <https://www.tutorialspoint.com/javafx/>

TutorialsTeacher. (2024). *JDBC – Java database connectivity overview*. Retrieved from <https://www.tutorialsteacher.com/jdbc>

W3Schools. (2024). *SQL tutorial – MySQL commands and queries*. Retrieved from <https://www.w3schools.com/sql/>

# Screenshots:

A screenshot of a computer

AI-generated content may be incorrect.

Login Screen

A screenshot of a computer

AI-generated content may be incorrect.

Register User

A screenshot of a computer

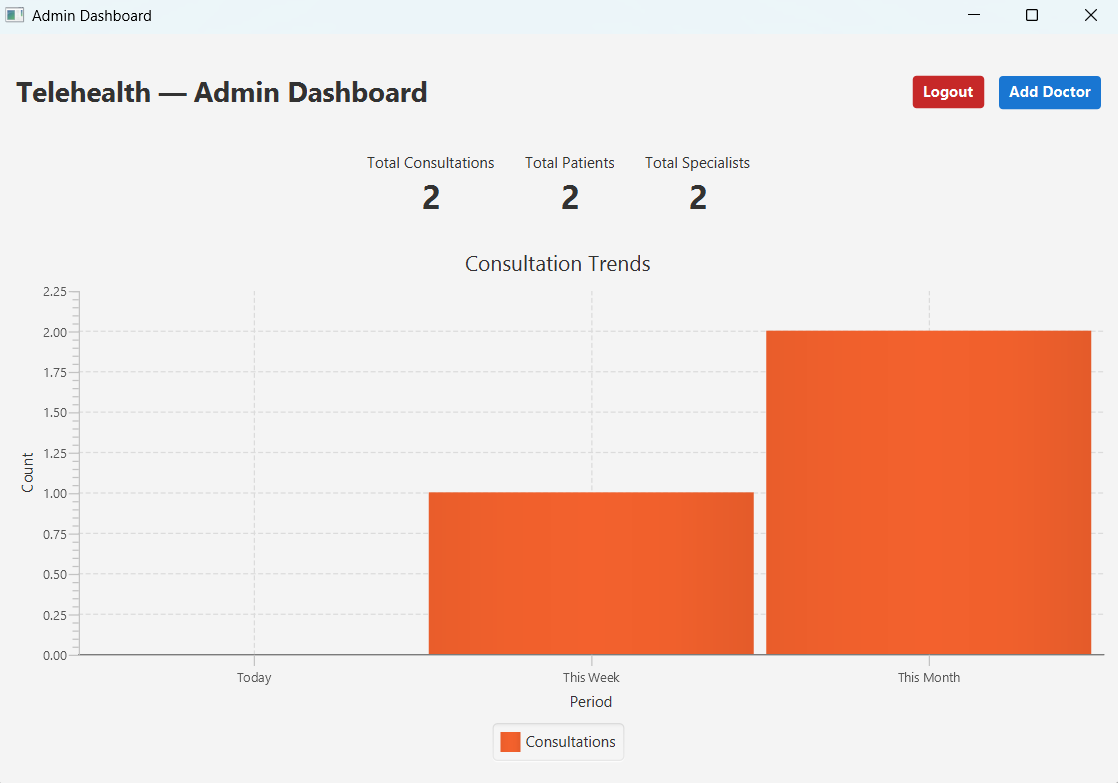
AI-generated content may be incorrect.

Patient Dashboard

A screenshot of a computer

AI-generated content may be incorrect.

Doctor dashboard



Admin login

A screenshot of a computer

AI-generated content may be incorrect.

Add Doctor

## Video Presentation Link

<https://echo360.net.au/media/dd24bf84-1bb7-4910-9521-c41ca6c55220/h5p>

## GitHub Link

<https://github.com/pketul2212/COIT20258-Assessment3>