**Knowledge Based System for Unani Medicine Formulations**



Session: 2021 - 2025

**Submitted by:**

Samee ul Rehman 2021-CS-116

Ali Hassan 2021-CS-148

Huzaifa Mumtaz 2021-CS-164

**Supervised by:**

Dr. Talha Waheed

**Co-Superviser:**

Dr. Samyaan Wahla

Department of Computer Science

**University of Engineering and Technology**

**Lahore Pakistan**

**Acknowledgments**

First and foremost, I express my gratitude to Allah almighty who give us strength and patience to work on this project and to do our research on Unani medicine and its formulations process.

After that, I would like to extend my appreciation our Supervisor Dr. Talha Waheed who helped us a lot and guide us properly which enhanced our knowledge and help out in completing this research. I also thank them for their proper guidance, valuable feedback and continuous encouragement which played a significant role throughout this journey. Their insight on Unani medicines and their formulation process help us to know that how the product prepared from natural ingredients and resources heal the bode of human being.

My sincere thanks go to the Chairman and Committee of the FYP, the Faculty and staff of Computer Science, and the respected Chairman Dr. Usman Ghani who always encouraged us and provided relevant resources and guidance throughout our FYP and degree journey, allowing us to grow academically and professionally.

Special thanks to my family and friends for their unwavering moral support, patience, and motivation throughout the course of my research.

Lastly, I acknowledge the use of scholarly literature, research databases, and digital libraries that contributed significantly to the formulation and validation of the knowledge-based system proposed in this thesis.

This research would not have been possible without the contributions and support of all the aforementioned individuals and institutions.

**Declaration**

Declaration We certify that the thesis is unique and original, apart from the explicitly cited sources, and that neither the entire thesis nor any individual chapters have been submitted previously for either the same bachelor’s degree or a different bachelor, or master’s degree. We also recognize and acknowledge that we have read and comprehended The University of Engineering and Technology, Lahore’s rules on dealing with academic dishonesty by students as well as its rules on student discipline.

Samee ul Rehman

Signed: Date:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ali Hassan

Signed: Date:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Huzaifa Mumtaz

Signed: Date:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Department of Computer Science

**University of Engineering and Technology**

**Lahore Pakistan**

**Knowledge Based System of Unani Medicine Formulations**

Submitted to the faculty of the Computer Science Department of the University of Engineering and Technology Lahore in partial fulfillment of the requirements for the Degree of Bachelor of Science in Computer Science.

**Internal Examiner** **External Examiner**

Signature: Signature:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Name: Name:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Designation: Designation:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Chairman** **Dean**

Signature: Signature:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. Dr. M. Usman Ghani Khan Prof. Dr. M. Shoaib

Department of Computer Science

**University of Engineering and Technology**

**Lahore Pakistan**

**Table of Contents**

[Chapter 1: Introduction 8](#_Toc9973)

[1.1 Project Background 8](#_Toc31785)

[1.2 Problem Statement 9](#_Toc8407)

[1.3 Objectives 10](#_Toc2720)

[1.4 Scope and Limitations The project required an absolute definition of its scope and boundaries for planning reasonable milestones and managing user expectations. 11](#_Toc14264)

[Scope 11](#_Toc23362)

[Limitations 12](#_Toc29088)

[1.5 Methodology Overview 13](#_Toc26437)

[1.6 Report Organization 13](#_Toc26890)

[1.7 Final Reflections 14](#_Toc21709)

[Chapter 2: Software Requirements Specification (SRS) 15](#_Toc12988)

[2.1 Introduction 15](#_Toc6132)

[2.1.1 Purpose 15](#_Toc813)

[2.1.2 Intended Audience 16](#_Toc26897)

[2.1.3 Definitions, Acronyms, and Abbreviations 16](#_Toc6184)

[2.2 Overall Description 17](#_Toc8781)

[2.2.1 Product Perspective 17](#_Toc20630)

[2.2.2 User Characteristics 17](#_Toc26305)

[2.2.3 Assumptions and Dependencies 18](#_Toc27544)

[Assumptions 18](#_Toc18363)

[Dependencies 18](#_Toc29006)

[2.3 Specific Requirements 19](#_Toc209)

[2.3.1 Functional Requirements 19](#_Toc6915)

[User Authentication and Registration 19](#_Toc27118)

[Search and Consultation 19](#_Toc32152)

[Admin Panel 19](#_Toc30343)

[Error Handling and Validation 20](#_Toc4193)

[2.3.2 Non-Functional Requirements 20](#_Toc5004)

[Usability 20](#_Toc15316)

[Performance 20](#_Toc469)

[Security 20](#_Toc14004)

[Maintainability 20](#_Toc31011)

[Reliability 21](#_Toc28067)

[2.3.3 Interface Requirements 21](#_Toc32742)

[User Interface (UI) 21](#_Toc20549)

[API Interface 21](#_Toc1635)

[Chapter 3: System Analysis and Requirements 23](#_Toc30796)

[3.1 Introduction 23](#_Toc2039)

[3.2 System Overview 23](#_Toc560)

[3.3 Existing Systems and Limitations 24](#_Toc19338)

[3.3.1 Scattered Resources 24](#_Toc27743)

[3.3.2 No Centralized Database 24](#_Toc25704)

[3.3.3 Poor User Experience 24](#_Toc14626)

[3.3.4 Trust Deficit 24](#_Toc24351)

[3.4 Proposed System 25](#_Toc12440)

[3.4.1 Organized Knowledge Base 25](#_Toc8723)

[3.4.2 Responsive, Modern Interface 25](#_Toc19298)

[3.4.3 Secure Access 25](#_Toc2019)

[3.4.4 Fast Search and Retrieval 25](#_Toc21509)

[3.4.5 Admin Management 25](#_Toc10716)

[3.5 System Requirements 25](#_Toc31758)

[3.5.1 Functional Requirements 25](#_Toc17157)

[3.5.2 Non-Functional Requirements 26](#_Toc19307)

[3.6 System Models 27](#_Toc6281)

[3.6.1 Use Case Diagram 28](#_Toc17320)

[3.6.2 Sequence Diagram 28](#_Toc31381)

[3.6.3 System Architecture Diagram 30](#_Toc7655)

[3.7 Database Design 31](#_Toc18645)

[3.7.1 Entity-Relationship (ER) Diagram 31](#_Toc465)

[3.7.2 Table Structures 32](#_Toc59)

[3.8 Risk Analysis 33](#_Toc17617)

[3.9 Summary 34](#_Toc17343)

[Chapter 4: Implementation 35](#_Toc31827)

[4.1 Introduction 35](#_Toc8233)

[4.2 Development Tools and Technologies 35](#_Toc17125)

[4.2.1 Visual Studio Code (VS Code) 35](#_Toc2856)

[4.2.2 Vue.js (Frontend) 36](#_Toc20325)

[4.2.3 Node.js and Express.js (Backend) 37](#_Toc24654)

[4.2.4 SQL Server (Database) 37](#_Toc13989)

[4.2.5 Postman (API Testing) 38](#_Toc29341)

[4.3 System Modules 38](#_Toc13099)

[4.3.1 Authentication Module 39](#_Toc16332)

[4.3.2 Search Module 39](#_Toc27641)

[4.3.3 Symptom Consultation Module 40](#_Toc31346)

[4.3.4 Admin Panel Module 40](#_Toc14973)

[4.3.5 API Communication Module 41](#_Toc2733)

[4.4 Key Implementation Decisions 41](#_Toc12569)

[4.4.1 Token Expiry for Sessions 41](#_Toc24848)

[4.4.2 Input Validation at Both Frontend and Backend 41](#_Toc28393)

[4.4.3 Lazy Loading in Vue.js 42](#_Toc5585)

[4.4.4 Standardized API Responses 42](#_Toc29356)

[4.4.5 Frontend Responsiveness 43](#_Toc24207)

[4.5 Sample Screenshots and UX Highlights 43](#_Toc28155)

[4.5.1 Homepage 43](#_Toc30507)

[4.5.2 Login and Registration Pages 43](#_Toc25394)

[4.5.3 Drug Search Results 44](#_Toc16008)

[4.5.5 Admin Dashboard 44](#_Toc27579)

[4.6 Reflections on Implementation 44](#_Toc3630)

[Chapter 5: Testing 46](#_Toc12703)

[5.1 Introduction 46](#_Toc19492)

[5.2 Testing Strategy 46](#_Toc15992)

[5.3 Types of Testing Conducted 47](#_Toc28801)

[5.3.1 Unit Testing 47](#_Toc4936)

[5.3.2 Integration Testing 48](#_Toc17529)

[5.3.3 System Testing 49](#_Toc22468)

[5.3.4 User Acceptance Testing (UAT) 50](#_Toc3834)

[5.4 Expanded Sample Test Cases 51](#_Toc9475)

[5.5 Bug Tracker and Issues Log 53](#_Toc17175)

[5.6 Testing Tools and Techniques Used 53](#_Toc20160)

[5.7 Expanded Reflections on Testing 54](#_Toc9930)

[Chapter 6: Deployment and User Manual 56](#_Toc26506)

[6.1 Introduction 56](#_Toc23773)

[6.2 Deployment Strategy 56](#_Toc24474)

[6.3 Backend Deployment (Node.js + Express.js) 57](#_Toc21237)

[6.3.1 Backend Preparation 57](#_Toc23329)

[6.3.2 Backend Hosting 58](#_Toc23826)

[6.3.3 Challenges Faced During Backend Deployment 58](#_Toc1107)

[6.4 Frontend Deployment (Vue.js) 59](#_Toc20998)

[6.4.1 Frontend Preparation 59](#_Toc32377)

[6.4.2 Frontend Hosting 59](#_Toc22705)

[6.4.3 Challenges Faced During Frontend Deployment 60](#_Toc2758)

[6.5 Database Deployment (SQL Server + SSMS) 60](#_Toc21504)

[6.5.1 SQL Server Setup 60](#_Toc13523)

[6.5.2 Data Migration 61](#_Toc3075)

[6.5.3 Challenges in Database Deployment 61](#_Toc19430)

[6.6 Post-Deployment Testing 61](#_Toc25751)

[6.7 User Manual 62](#_Toc22376)

[6.7.1 Accessing the System 62](#_Toc13939)

[6.7.2 Registration 62](#_Toc26695)

[6.7.3 Login 62](#_Toc17824)

[6.7.4 Searching Drugs 63](#_Toc30217)

[6.7.5 Symptom Consultation 63](#_Toc26783)

[6.7.6 Admin Dashboard (Admins Only) 63](#_Toc11744)

[6.7.7 Logging Out 63](#_Toc19911)

[6.8 Maintenance Guidelines 63](#_Toc2819)

[6.8.1 Backend Maintenance 64](#_Toc30891)

[6.8.2 Frontend Maintenance 64](#_Toc2831)

[6.8.3 Database Maintenance 64](#_Toc8064)

[6.8.4 Security Maintenance 64](#_Toc9797)

[6.9 Reflections on Deployment 64](#_Toc4314)

[Chapter 7: Results, Evaluation, and Conclusion 66](#_Toc13021)

[7.1 Introduction 66](#_Toc19520)

[7.2 Summary of Achievements 66](#_Toc21445)

[7.2.1 Technical Achievements 66](#_Toc3167)

[7.2.2 Functional Achievements 67](#_Toc22148)

[7.2.3 User-Centric Achievements 68](#_Toc14316)

[7.3 Challenges Faced 68](#_Toc16671)

[7.3.1 Technical Challenges 69](#_Toc5744)

[7.3.1.1 Backend and Frontend Synchronization 69](#_Toc6190)

[7.3.1.2 CORS Issues 69](#_Toc20626)

[7.3.1.3 SQL Query Performance 69](#_Toc26239)

[7.3.2 Management and Planning Challenges 69](#_Toc16285)

[7.3.3 Resource Limitations 70](#_Toc6912)

[7.4 Lessons Learned 70](#_Toc28247)

[7.4.1 Lesson: Testing Must Start Early 70](#_Toc14018)

[7.4.2 Lesson: Assume Nothing About Users 70](#_Toc32096)

[7.4.3 Lesson: Simplicity Wins 70](#_Toc21591)

[7.4.4 Lesson: Small Errors Cause Big Problems 71](#_Toc28592)

[7.5 Future Enhancements 71](#_Toc6872)

[7.5.1 Move to Scalable Cloud Hosting 71](#_Toc18421)

[7.5.2 AI-Powered Consultations 71](#_Toc593)

[7.5.3 Mobile Application Development 72](#_Toc26069)

[7.5.4 Multilingual Support 72](#_Toc15469)

[7.5.5 Practitioner Collaboration Platform 72](#_Toc19051)

[7.6 Final Reflections 72](#_Toc27536)

[7.7 Conclusion 73](#_Toc25971)

[References 74](#_Toc25544)

**List of Figures**

[Figure 1 29](#_Toc20003)

[Figure 2 30](#_Toc10740)

[Figure 3 32](#_Toc30450)

[Figure 4 33](#_Toc2094)

**List of Tables**

[Table 1 37](#_Toc29459)

[Table 2 56](#_Toc1066)

[Table 3 56](#_Toc18870)

## **Chapter 1: Introduction**

## **1.1 Project Background**

Human civilization has always depended on medicine which develops through the combination of observation and experience and scientific inquiry. Traditional medicine includes Unani as an established approach to healing which adapts ancient Greek medical concepts through extended development by Arab and Persian and South Asian scholars during successive centuries. **Unani medicine** follows a body humoral theory based on blood and phlegm and yellow bile and black bile which focuses on complete body wellness instead of treating individual symptoms. Unani medicine utilizes plant-based and mineral compounds and dietary treatments to achieve long-term health recovery instead of providing temporary solutions.

The widespread use of Unani medicine in India as well as Pakistan and throughout the Middle East fails to match its historical significance because it finds limited representation in contemporary digital platforms. Modern Unani medicine remains primarily confined to old manuscripts and books along with the knowledge passed down through oral traditions of experienced practitioners while allopathic and Ayurvedic medicine have extensively adopted mobile apps and online databases and research portals. The current accessibility gap creates a significant threat because students and patients will avoid systems including Unani when they need information or services through preferred digital means.

We recognized the digital disparity in Unani formulations which led us to develop a platform that would digitize their organization to make them searchable and understandable for a broad user base. Our team developed the Knowledge-Based System (KBS) for Unani Medicine Formulations due to such requirements. The technical project functions as a cultural preservation initiative dedicated to preserving vital medical heritage components. Vue.js workflows with Node.js server technologies and SQL Server databases and Postman testing functions combined to create an information system which delivers Unani wisdom through modern user-friendly speed requirements.

We pursued this project with the dual purpose of both academic contribution and implementing modern digital practices for preserving ancient medical wisdom.

## **1.2 Problem Statement**

Unani medicine has stood the test of time yet faces numerous survival issues in modern digital information systems.

We identified major issues within our initial investigation phase:

The major obstacle to **knowledge distribution** exists as the first challenge. Due to its decentralized nature Unani formulation knowledge exists only in various historical textbooks along with manuscripts and elderly learned practitioners remember its contents.  
Students, researchers, and health enthusiasts often struggle to find reliable, consolidated resources online. When information is available, it is either incomplete, lacks credibility, or is locked behind physical access limitations like specific university libraries or regional archives.

**The inability of modern users to access Unani knowledge** stands as the second main issue. Current students and healthcare workers together with ordinary audiences want instant and convenient digital interfaces for accessing information. Most people cannot access Unani knowledge because the lack of a mobile-friendly web-based platform bars them from accessing it especially when they typically expect quick answers through Googling.

Thirdly, there is **a real danger of knowledge loss**.  
The disappearance of essential insights and formulations from Unani medicine becomes likely because senior practitioners terminate their practice while deteriorating manuscripts go without proper maintenance. The destruction of alternative medical strategies may occur because future generations will not have access to preserved and platform-ready traditional medical methods.

Finally, there is **a trust deficit**.  
  
Users who cannot access credible digital platforms containing Unani medicine information might think it is outdated or irrelevant, leading them to avoid using it.

The combination of these issues creates a serious problem because Unani medicine faces potential disappearance from public awareness when there is no strategic digitization alongside proper organization.

This Knowledge-Based System created **reliable search functions** which combine **x** for Unani knowledge preservation throughout the 21st century.

## **1.3 Objectives**

The project objectives served both as precise solutions for the identified issues and direction for developing meaningful results.

* **Digitization and Structuring of Knowledge**:  
  One of the primary goals was to collect authentic Unani drug information and formulations and organize them into a structured, relational database. This would allow for easy retrieval, filtering, and updating of information.
* **Development of a User-Friendly Web Platform**:  
  Using Vue.js, the project aimed to create an intuitive and responsive frontend that would appeal to both technical and non-technical users. Ease of navigation, clear search functionality, and mobile responsiveness were key design principles.
* **Secure Backend and Data Management**:  
  Node.js was used to build the backend, ensuring efficient API development, secure user authentication, and smooth management of database interactions. SQL Server was selected for its reliability, scalability, and powerful querying capabilities.
* **Testing and Validation**:  
  Postman and manual testing processes were employed throughout development to ensure that APIs, user actions, and data flows worked as expected under different scenarios.
* **Deployment and Maintenance Readiness**:  
  While the project initially focused on local and university-level deployment, the system was built with future readiness for broader deployment, should additional funding or institutional interest arise.

In summary, Social impact served as the main goal along with technological achievement when restoring the ancient system through formats that modern users find familiar and dependable.

## **1.4 Scope and Limitations** The project required an absolute definition of its scope and boundaries for planning reasonable milestones and managing user expectations.

### **Scope**

* **User Functionality**:  
  The system permits standard users to find drugs through their name or scientific name and their medicinal properties. Drugs administration receives formulation recommendations through keyword matching inputs.
* **Admin Functionality**:  
  The database management system enables administrators to connect securely to an admin dashboard to make additions, changes and removals of drugs and formulations while upholding database accuracy and relevancy.
* **Technology Stack**:  
  The system was developed using Vue.js for the frontend, Node.js for backend server logic, SQL Server for the relational database, and Postman for API testing and validation.
* **Security Measures**:  
  User authentication was implemented using JSON Web Tokens (JWT), ensuring secure login sessions and admin access control.
* **Responsiveness and Accessibility**:  
  The web platform is designed to be responsive, making it accessible across devices, from desktop monitors to smartphones.

### **Limitations**

* **Language Support**:  
  The current version supports English only. Expanding to languages like Urdu and Arabic is planned for future versions but was beyond this project’s initial scope.
* **Consultation Intelligence**:  
  Symptom consultations are based on simple keyword matching rather than advanced AI-based diagnostic systems.
* **Hosting and Deployment**:  
  Due to budget constraints, the system was deployed on limited infrastructure suitable for academic evaluation but not massive public usage.
* **Database Scale**:  
  While the system is scalable by design, the initial database includes a curated set of formulations rather than an exhaustive encyclopedia of Unani medicine.

By outlining both the possibilities and the boundaries of the system, we ensured that the project stayed focused and achievable within available resources.

## **1.5 Methodology Overview**

The system development employed a **hybrid Agile methodology** because of project dynamics together with requirements for adaptability.  
This approach blended iterative, flexible development cycles with structured milestones typical of the Waterfall model, ideal for academic timelines.

The major stages included:

* **Requirement Gathering**:  
  Initial research into Unani formulations, system expectations, and user personas shaped early planning documents.
* **System Design**:  
  Database schema design, API structure planning, and frontend wireframes were drafted to visualize the system architecture.
* **Incremental Development**:  
  Two-week sprints focused on achieving specific deliverables, such as completing the login module, building search APIs, or integrating symptom consultations.
* **Testing and Feedback**:  
  After each sprint, modules were tested using Postman and manual UI testing. Immediate feedback loops helped catch and correct issues early.
* **Documentation and Finalization**:  
  Throughout the project, thorough documentation was maintained for easier handover, evaluation, and future extension.

This flexible yet structured approach allowed the team to respond quickly to challenges while maintaining overall project direction and quality.

## **1.6 Report Organization**

The structure of this report is designed to logically guide readers through the project journey:

* **Chapter 1: Introduction** — Background, Problem Statement, Objectives, Scope, Methodology Overview.
* **Chapter 2: Software Requirements Specification (SRS)** — Detailed requirements of the system.
* **Chapter 3: System Analysis and Requirements** — In-depth use case and system modeling.
* **Chapter 4: Implementation** — Tools used, module descriptions, key implementation details.
* **Chapter 5: Testing** — Testing strategies, test cases, results.
* **Chapter 6: Deployment and User Manual** — How the system was deployed and instructions for users.
* **Chapter 7: Results, Evaluation, and Conclusion** — Achievements, challenges faced, lessons learned, and future work.

This logical organization ensures clarity, coherence, and easy navigation for academic evaluators and future readers.

## **1.7 Final Reflections**

Technical excellence united with cultural accountability shaped the development process of the Knowledge-Based System for Unani Medicine Formulations. This project carried the mission to safeguard centuries of medical knowledge through development of a system which both contemporary users and dependable data access could utilize.

During the project work we tested and improved our technological abilities yet our main learning responsibility dealt with combining established traditions with emerging technological advances.

Through this project we contribute modestly to the preservation of Unani medicine in the digital infrastructure which may emerge in the future.

# **Chapter 2: Software Requirements Specification (SRS)**

## **2.1 Introduction**

An SRS document specifies all the necessary components to create, develop and support the KBS system for Unani Medicine Formulations.

The SRS provides foundational directions to developers, testers and academic supervisors and future maintainers to implement the system.

A strong SRS is more than just a formality; it ensures that all team members work with a shared understanding of expectations, limitations, and success criteria. Without such documentation, projects risk scope creep, misunderstandings, and avoidable rework.

This chapter outlines the purpose, intended audience, product perspective, user characteristics, assumptions, dependencies, specific functional and non-functional requirements, and necessary system interfaces for the KBS.

## **2.1.1 Purpose**

The main purpose of this SRS is to:

* Clearly define the goals and expected behavior of the Knowledge-Based System.
* Provide a single, comprehensive reference document for developers and stakeholders throughout the system's lifecycle.
* Identify both what the system must do (functional requirements) and how it must perform (non-functional requirements).
* Highlight system constraints, assumptions, and dependencies to ensure realistic planning.
* Support quality assurance efforts by providing a basis for test case design and evaluation.

By laying down precise requirements early, this document minimizes the risk of project deviation and ensures that the final product remains aligned with the original academic and user expectations.

## **2.1.2 Intended Audience**

The intended readers of this SRS document include:

* **Software Developers**: To build the frontend, backend, and database components according to specifications.
* **Quality Assurance Testers**: To design comprehensive test cases based on defined system functionalities.
* **Academic Supervisors and Evaluators**: To assess the project's depth, relevance, and technical execution.
* **Future Maintainers and Developers**: To extend or modify the system, using this document as a reliable reference.
* **Healthcare Researchers and Students**: Interested in understanding the system’s capabilities to access Unani medicine knowledge.

Efforts have been made to write the SRS in clear, concise language that balances technical accuracy with accessibility for non-developers.

## **2.1.3 Definitions, Acronyms, and Abbreviations**

* **KBS**: Knowledge-Based System
* **UI**: User Interface
* **API**: Application Programming Interface
* **JWT**: JSON Web Token
* **CRUD**: Create, Read, Update, Delete
* **SQL**: Structured Query Language
* **SSMS**: SQL Server Management Studio
* **VS Code**: Visual Studio Code (IDE)

Clarifying terminology ensures consistent understanding across different team members and reviewers.

# **2.2 Overall Description**

## **2.2.1 Product Perspective**

The Knowledge-Based System is an independent, standalone web application developed to digitize, organize, and present Unani medicinal formulations.  
It does not build on any pre-existing software but was designed from scratch to address specific accessibility and preservation challenges faced by Unani medicine.

The system integrates:

* **Vue.js Frontend**: For user interaction and presentation logic.
* **Node.js Backend (Express Framework)**: For business logic, authentication, and API management.
* **SQL Server Database**: For structured, relational storage of drugs, formulations, symptoms, and user information.
* **Postman**: For API development and testing.
* **VS Code**: As the development environment.

The overall design ensures modularity, where frontend, backend, and database layers communicate through RESTful APIs while remaining loosely coupled for easier maintenance and scaling.

## **2.2.2 User Characteristics**

The system is designed for three primary categories of users:

* **General Users**:  
  Health enthusiasts, students, or researchers seeking information about Unani drugs and formulations. They require a clean, intuitive interface with minimal technical complexity.
* **Admin Users**:  
  Authorized individuals responsible for managing the database entries. They require advanced access to CRUD operations but within a secure, user-friendly dashboard.
* **Supervisors/Researchers**:  
  Individuals interested in evaluating, studying, or expanding the database, who would primarily interact with search and consultation features.

All users are assumed to have basic internet browsing skills but are not required to have any prior technical expertise in database systems or web development.

## **2.2.3 Assumptions and Dependencies**

### **Assumptions**

* Users will access the platform via modern web browsers like Chrome, Firefox, Safari, or Edge.
* Internet connectivity will be stable enough to support uninterrupted browsing and server interactions.
* Admin users will be responsible for maintaining the integrity and relevance of the drug and formulation entries.
* Authentication via JWT will be secure enough for academic usage.

### **Dependencies**

* **Frontend-Backend Communication**: RESTful APIs must remain operational and standardized to maintain smooth user experience.
* **Database Availability**: The SQL Server database must be hosted and maintained properly to ensure data integrity and uptime.
* **Node.js Server Uptime**: Backend APIs must remain functional and available to support frontend operations.
* **Email Services**: Required for password recovery functionality (e.g., SMTP server).

Understanding these assumptions and dependencies is vital for predicting system behavior under real-world conditions.

# **2.3 Specific Requirements**

# **2.3.1 Functional Requirements**

Functional requirements define the core tasks that the system must support.

### **User Authentication and Registration**

* New users must be able to register by providing a valid email and password.
* Users must be able to log in securely with JWT-based session management.
* Users must be able to reset forgotten passwords through email verification.

### **Search and Consultation**

* Users must be able to search for drugs by:
  + Name
  + Scientific name
  + Medicinal properties
* Users must be able to input symptoms and receive formulation suggestions based on keyword matching.

### **Admin Panel**

* Admin users must be able to:
  + Add new drug entries.
  + Edit existing drug entries.
  + Delete outdated or incorrect formulations.
  + View lists of users (future enhancement).

### **Error Handling and Validation**

* All user inputs must be validated on both frontend and backend sides.
* Clear error messages must guide users when invalid inputs are detected.

## **2.3.2 Non-Functional Requirements**

Non-functional requirements specify how the system must behave, beyond specific functionalities.

### **Usability**

* The user interface must be clean, modern, and responsive, ensuring accessibility on desktops, tablets, and smartphones.
* Navigation must be intuitive, minimizing the number of clicks required to perform core tasks.

### **Performance**

* API responses must be delivered within 2 seconds under normal conditions.
* Search results must display within 3 seconds on a standard 4G connection.

### **Security**

* Passwords must be encrypted before storage.
* JWT tokens must be securely generated and validated.
* API endpoints must prevent unauthorized access to sensitive operations.

### **Maintainability**

* The codebase should use modular components for scalability.
* RESTful APIs must adhere to standard best practices for clarity and reusability.

### **Reliability**

* The system should target 99% uptime (assuming academic hosting conditions).
* Regular data backups should be performed (manual initially).

## **2.3.3 Interface Requirements**

The system involves two major interface groups:

### **User Interface (UI)**

* Built using Vue.js.
* Features include a homepage, search page, consultation page, registration/login forms, and an admin dashboard.
* Responsive design must be compatible across major browsers and devices.

### **API Interface**

* All interactions between the frontend and backend must use RESTful APIs.
* API endpoints must return data in JSON format.
* Secure endpoints (e.g., admin operations) must require JWT authentication.

Example API Endpoints:

* POST /auth/register
* POST /auth/login
* POST /auth/forgot-password
* GET /drugs
* GET /formulations
* POST /consultations
* POST /admin/add-drug
* PUT /admin/edit-drug/:id
* DELETE /admin/delete-drug/:id

The consistency of API structures and error handling is critical to system reliability and developer ease.

# **Chapter 3: System Analysis and Requirements**

## **3.1 Introduction**

Before jumping into development, thorough system analysis is essential.  
It allows a clear understanding of the existing problems, user expectations, system environment, and technical feasibility.  
For the Knowledge-Based System (KBS) for Unani Medicine Formulations, system analysis was a bridge connecting conceptual ideas with realistic, executable plans.

This chapter explores the background study, identifies the gaps in current systems, elaborates on proposed solutions, analyzes user and system requirements, presents modeling diagrams, discusses database design, highlights potential risks, and concludes with reflections on how these preparations shaped the project's success.

## **3.2 System Overview**

The Knowledge-Based System aims to digitally preserve, organize, and offer easy access to authentic Unani medicinal knowledge.  
Users — whether researchers, students, or health enthusiasts — can search for traditional formulations based on drug names, properties, or symptoms. Admins, meanwhile, can manage and curate content.

Key actors:

* General User (searches and consults)
* Admin User (manages content)
* Backend Server (processes requests, manages authentication)
* SQL Database (stores drugs, symptoms, formulations)

In short, the system seeks to modernize ancient medical practices into accessible, fast, and user-friendly digital experiences without compromising traditional authenticity.

## **3.3 Existing Systems and Limitations**

During the preliminary research, it became evident that there were few, if any, credible Unani systems online.  
Several issues were observed:

### **3.3.1 Scattered Resources**

Most Unani knowledge exists in old printed books, handwritten notes, or within practitioners' memory.  
Even when available digitally, it is spread across random blog posts, outdated PDFs, and non-searchable scanned documents.

### **3.3.2 No Centralized Database**

There is no structured, relational database that connects drugs, properties, symptoms, and formulations in a user-friendly manner.  
Users often have to sift through long text paragraphs to extract tiny bits of information.

### **3.3.3 Poor User Experience**

Existing medical portals, if any, are often designed poorly — cluttered pages, unresponsive layouts, slow search functionalities, and lack of mobile optimization alienate modern users.

### **3.3.4 Trust Deficit**

Because there is no authoritative, professionally maintained system for Unani medicine online, users often doubt the reliability of the information.

Thus, the absence of a credible, easy-to-use, accessible platform inspired the development of a dedicated KBS for Unani Medicine Formulations.

## **3.4 Proposed System**

Our proposed system addresses these gaps head-on with the following key features:

### **3.4.1 Organized Knowledge Base**

Data is properly categorized into Drugs, Symptoms, and Formulations. Users can search by multiple attributes and navigate seamlessly.

### **3.4.2 Responsive, Modern Interface**

Using Vue.js, we crafted a lightweight, elegant, and mobile-responsive frontend that works fluidly across devices.

### **3.4.3 Secure Access**

JWT-based authentication secures user sessions and admin functionalities.

### **3.4.4 Fast Search and Retrieval**

Indexed SQL tables ensure that users get search results within seconds, even with thousands of records.

### **3.4.5 Admin Management**

Admins can add, edit, or remove formulations without needing technical knowledge of databases — all handled via a clean dashboard.

## **3.5 System Requirements**

A project cannot succeed without a precise understanding of what is expected.  
System requirements were divided into functional and non-functional categories.

### **3.5.1 Functional Requirements**

**User Registration and Authentication**

* Allow users to register, login, and recover passwords securely.

**Drug Search**

* Users can search for drugs by name, scientific name, or properties.

**Symptom Consultation**

* Users input symptoms and receive matching formulations.

**Admin Management**

* Admin users can add, update, or delete drugs and formulations.

**API Communication**

* Frontend and backend interact seamlessly via standardized REST APIs.

**Error Handling**

* Display clear, friendly error messages for invalid operations or server failures.

### **3.5.2 Non-Functional Requirements**

**Performance**

* Search results and consultations must load within 3 seconds on a 4G network.

**Scalability**

* The system must support scaling the database and server load if user base grows.

**Security**

* Passwords stored as hashed values.
* JWT sessions expire after a reasonable time.

**Maintainability**

* Modular coding structure for easy updates and feature expansions.

**Responsiveness**

* UI must adjust gracefully across devices — desktop, tablet, and smartphones.

**Availability**

* Target 99% uptime in an academic hosting environment.

## **3.6 System Models**

System modeling visually captures how the users, system, and data interact.

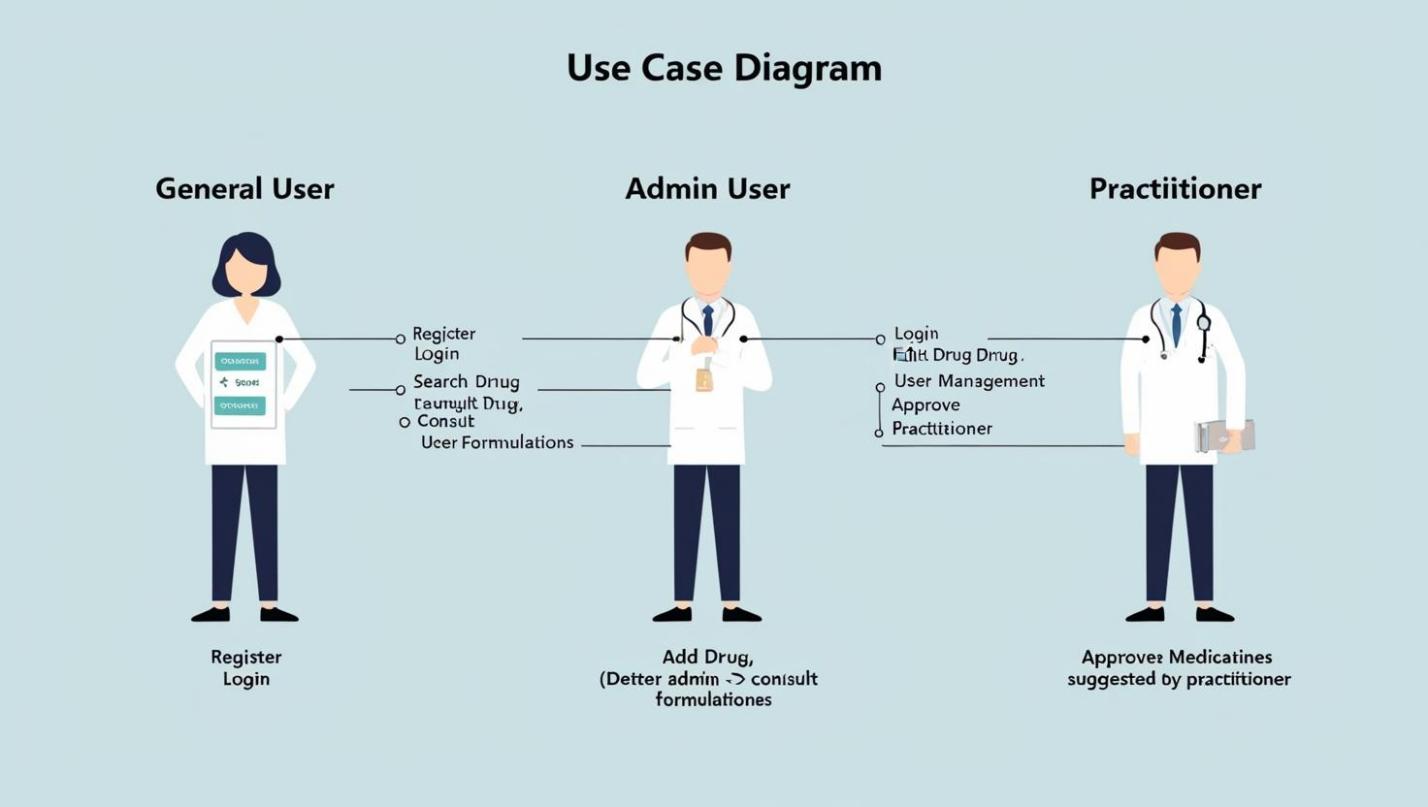


Figure 1

### **3.6.1 Use Case Diagram**

Actors:

* **General User**
* **Admin User**
* **Practitioner**

Use Cases:

* Register
* Login
* Search Drug
* Consult Formulation
* Add/Edit/Delete Drug (Admin only)

(A graphical Use Case diagram connects these actors with their respective system interactions.)

### **3.6.2 Sequence Diagram**

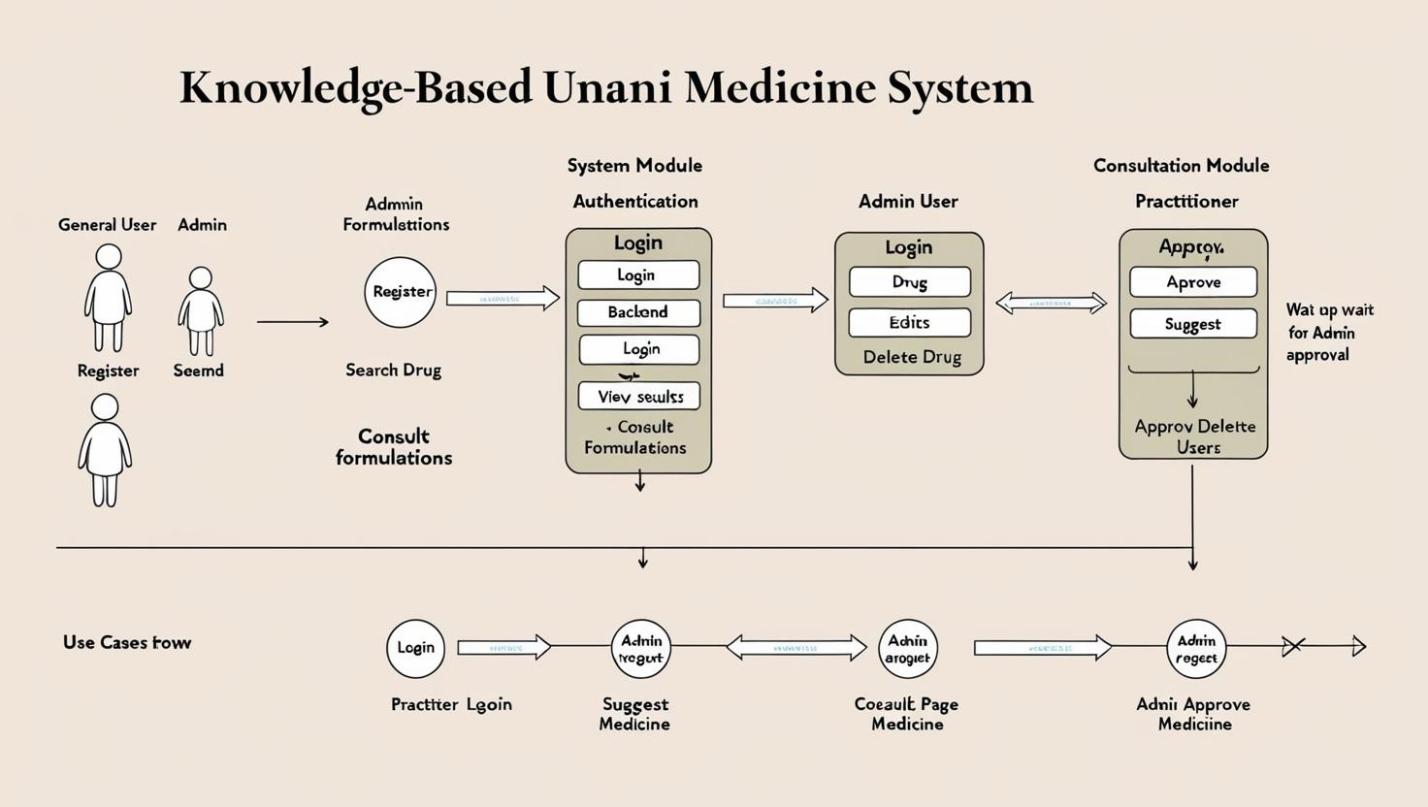


Figure 2

The Knowledge-Based Unani Medicine System is geared to assist users in consulting Unani drug formulations and handling medicine suggestions through an organized approval mechanism. The system accommodates **three primary user** categories: General Users, Admins and Practitioners. General Users are able to register, sign in, look up drugs, and consult formulation. Admins handle backend functionality like approving new practitioners, updating drug details, and removing stale records. Practitioners will be able to modify already stored medicines and can add new formulations but with the approval of admin. The system is separated into **three main modules**: **Authentication** (managing user login and access to formulations), **Admin User Management** (for editing and deleting drug records), and the **Consultation Module** (where practitioners recommend medicines). Practitioners can recommend new medicines, but the recommendations are pending approval by an Admin. Use case flow identifies a cycle where practitioners propose medicines, admins review and validate them before accepting them finally, maintaining data integrity and controlled handling of the Unani formulations database.

Example: **Login Sequence**

1. User submits login form.
2. Frontend sends API request to backend.
3. Backend verifies credentials with SQL database.
4. If valid, backend issues JWT token.
5. Frontend stores JWT token and redirects user to dashboard.

This sequence ensures secure, verified sessions.

### **3.6.3 System Architecture Diagram**

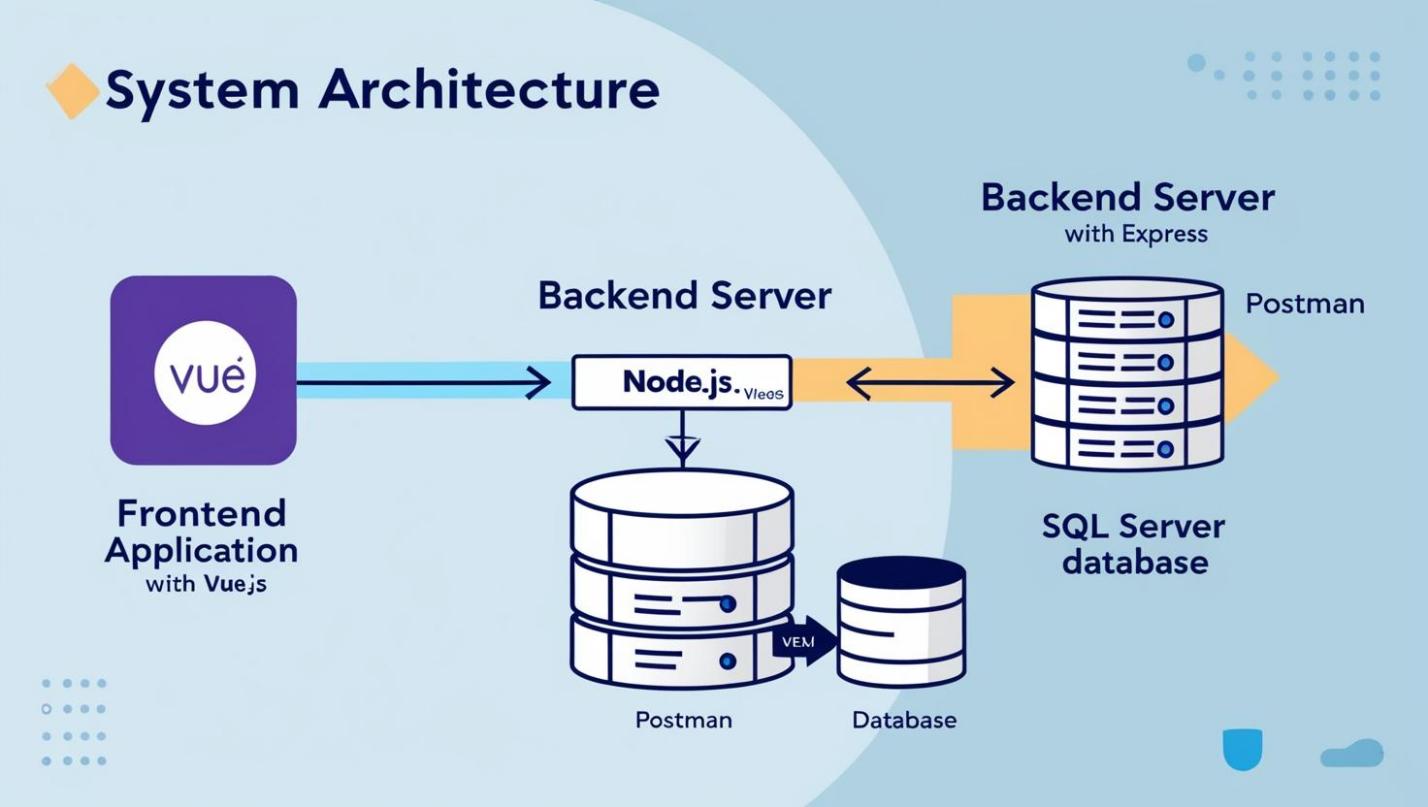


Figure 3

The architecture of the system shows the interaction between frontend application, backend server, and database. The Frontend Application is developed in Vue.js, which manages the user interface and user interactions. This frontend talks to the Backend Server, which is developed in Node.js with Express.js. Node.js is a middleware that processes user requests from the frontend, talks to the database, and returns the necessary responses back. Postman is utilized as an API testing tool to provide seamless communication between the client and server. The backend server further communicates with the SQL Server database to retrieve, insert, update, or delete data as needed. This design provides a structured and efficient information flow between users, servers, and the database while ensuring system reliability and performance.

* Users interact with frontend.
* Frontend interacts with backend APIs.
* Backend reads/writes to the SQL database.

## **3.7 Database Design**

Proper database architecture was critical for fast, reliable information retrieval.

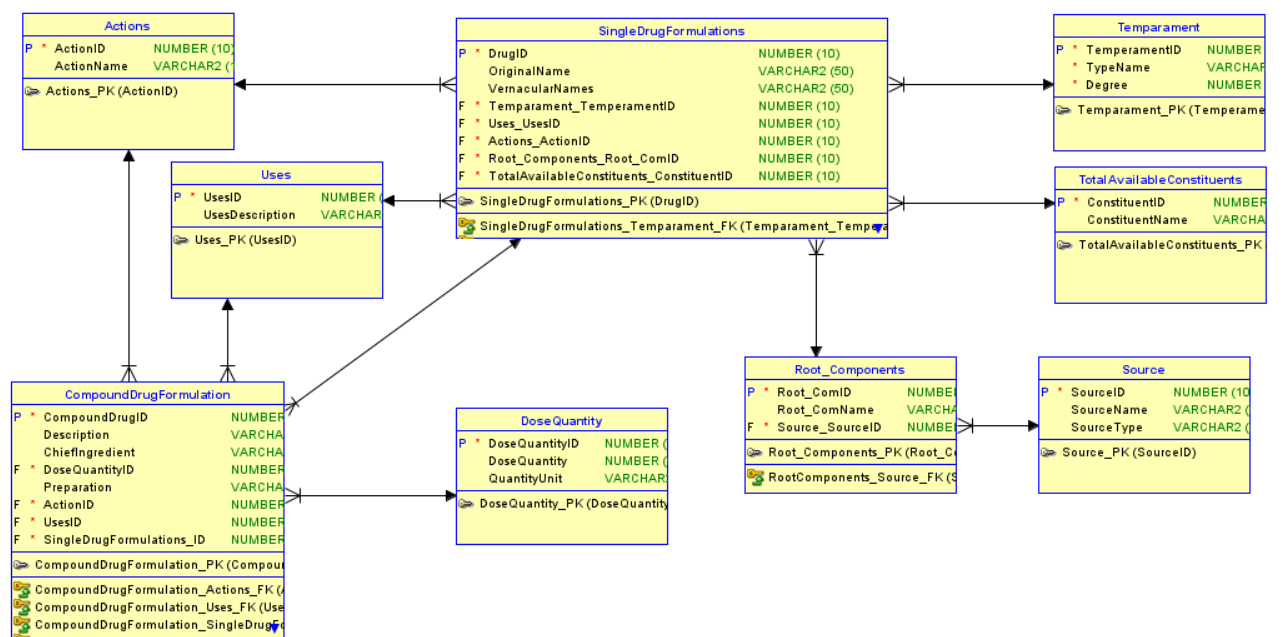


Figure 4

The schema of the Unani Medicine Formulation System database is structured so that it can effectively store and retrieve information regarding single and compound drug formulations. The core of this is the SingleDrugFormulations table, which houses information such as drug names, temperaments, uses, root components, related actions, and available constituents. It relates to support tables such as Actions, Uses, Temperament, Root\_Components, and TotalAvailableConstituents via foreign key relationships. Every drug formulation is traced back to some action, use case, and type of temperament. Root ingredients are stored in the Root\_Components table, which have their sources connected through the Source table. Compound drugs are stored independently in the CompoundDrugFormulation table, which has references to several dimensions such as dose quantity (DoseQuantity table), accompanying action, uses, and a related single drug formulation where needed. In total, the schema provides a normalized, structured, and flexible design to effectively handle complex medicinal information.

### **3.7.1 Entity-Relationship (ER) Diagram**

Entities:

* **Users** (user\_id, name, email, password, is\_admin)
* **Drugs** (drug\_id, name, scientific\_name, properties, description)
* **Symptoms** (symptom\_id, name)
* **Formulations** (formulation\_id, drug\_id, symptom\_id, preparation, usage\_instructions)

Relationships:

* One drug can treat multiple symptoms.
* One symptom can relate to multiple drugs.

This normalized structure avoids redundancy and optimizes query performance.

### **3.7.2 Table Structures**

**Users Table**

* user\_id (PK)
* email (unique)
* password\_hash
* is\_admin (boolean)

**Drugs Table**

* drug\_id (PK)
* name
* scientific\_name
* properties
* description

**Symptoms Table**

* symptom\_id (PK)
* name

**Formulations Table**

* formulation\_id (PK)
* drug\_id (FK)
* symptom\_id (FK)
* preparation
* usage\_instructions

## **3.8 Risk Analysis**

Risk assessment during early stages helped prepare mitigation strategies.

|  |  |  |  |
| --- | --- | --- | --- |
| **Risk** | **Impact** | **Likelihood** | **Mitigation Strategy** |
| CORS Errors | Medium | High | Test API headers early |
| SQL Injection | High | Medium | Sanitize and validate all inputs |
| Server Downtime | Medium | Medium | Implement backup hosting options |
| User Confusion | Medium | High | Keep UI simple and add tooltips |
| Loss of Data | High | Low | Weekly database backups |
| Unauthorized Admin Access | Critical | Low | Strict JWT and role-based access control |

Table 1

## **3.9 Summary**

System analysis with modeling created a solid base for the KBS project development. The outcome of a meticulous evaluation between system constraints, user requirements, technological capabilities and potential dangers resulted in an authentic and solid plan. The initial analysis greatly reduced development time through the prevention of misinterpretations and the neutralization of scope creep and the identification of logical reasons behind each programming decision. Understanding that the system would serve ancient medical knowledge to thousands of future users deepened my responsibility to focus on quality alongside security and usability during the entire development period.

# **Chapter 4: Implementation**

## **4.1 Introduction**

Reality meets theory at the time when implementation begins. The development phase of the Knowledge-Based System (KBS) for Unani Medicine Formulations began after extensive planning where requirements were gathered and system models were created. Building the system was more than writing code because it required thinking like an architect while optimizing resources and solving creative problems and testing without cease. The development environment and selected tools and technologies with modular design structure along with key implementation decisions and encountered challenges and system highlight screenshots and an evaluation of iterative development process are explained in detail in this chapter.

## **4.2 Development Tools and Technologies**

Choosing the right technology stack was pivotal for a successful, maintainable project.

### **4.2.1 Visual Studio Code (VS Code)**

Visual Studio Code emerged as the integrated development environment (IDE) of choice for the project.

Why VS Code?

* **Lightweight and Fast**: Easily handled large codebases without performance lags.
* **Highly Customizable**: Through extensions for Vue.js syntax, REST API testing, Git integration.
* **Built-in Terminal**: Allowed running backend servers, Node.js scripts, and npm commands directly without leaving the editor.
* **Version Control**: Git extensions simplified repository management, branch creation, and commit tracking.

Real-world benefit:  
 Using VS Code, frontend and backend codes were handled simultaneously in a single workspace, improving development efficiency and focus.

### **4.2.2 Vue.js (Frontend)**

Vue.js, a progressive JavaScript framework, was selected to build the user-facing frontend application.

Key strengths leveraged:

* **Component-Based Architecture**: Each page (Search, Consultation, Login, Admin Dashboard) was broken into manageable, reusable Vue components.
* **Reactive UI**: Vue’s reactive data binding ensured instant updates to user inputs, delivering a smooth browsing experience.
* **Vue Router**: Managed client-side routing, allowing seamless transitions between pages without full page reloads.
* **Flexibility**: Vue’s simplicity allowed fast learning curves and easy customization compared to heavier frameworks like Angular.

Real-world benefit:  
 By using Vue, even complicated functionalities like symptom consultation or dynamic search suggestions were handled elegantly.

### **4.2.3 Node.js and Express.js (Backend)**

The backend server was powered by Node.js (runtime) and Express.js (web framework).

Why this choice?

* **Non-blocking I/O**: Handled multiple concurrent API requests without server crashes.
* **Minimalist Framework**: Express.js provided just enough structure without overcomplicating server logic.
* **Rich Middleware Ecosystem**: Incorporated authentication, CORS handling, body parsing, and error handling quickly.

Real-world benefit:  
 Backend APIs were rapidly developed, tested, and integrated without unnecessary boilerplate, keeping server-side codebase lean and manageable.

### **4.2.4 SQL Server (Database)**

SQL Server was chosen as the database management system due to its robustness, scalability, and security features.

Benefits realized:

* **Structured Query Language (SQL)**: Provided precise, optimized querying.
* **Data Integrity**: Supported strong schema enforcement (primary keys, foreign keys, constraints).
* **Indexing**: Speeded up complex search operations significantly.
* **Backup and Recovery Tools**: SSMS allowed easy data export/import operations for backups.

Real-world benefit:  
 Queries like “Find formulations for symptoms X, Y, Z” returned results in seconds, even as data volume grew.

### **4.2.5 Postman (API Testing)**

Postman was indispensable during backend API testing phases.

Uses:

* **API Testing**: Simulated user requests for login, search, consultation, password reset.
* **Authentication Testing**: Managed JWT bearer tokens for secure endpoint testing.
* **Error Simulation**: Sent invalid data to verify error handling.
* **Chained Requests**: Tested multi-step processes like registration ➔ login ➔ drug search in one workflow.

Real-world benefit:  
 Postman detected API inconsistencies early, allowing rapid fixes before frontend integration.

## **4.3 System Modules**

The KBS was organized into several logical modules to promote separation of concerns, code reusability, and easier testing.

### **4.3.1 Authentication Module**

Handles user registration, login, and password recovery.

Implementation Highlights:

* Passwords were hashed using bcrypt.js before storage.
* JWTs issued after login included expiry times to enhance security.
* Forgot-password flow triggered password reset emails through SMTP server (local for now).

Challenges:

* Balancing security strictness and user experience simplicity during authentication design.

### **4.3.2 Search Module**

Allows users to search drugs based on name, scientific name, or properties.

Implementation Highlights:

* Frontend forms submitted keyword queries to backend APIs.
* Backend sanitized inputs and queried indexed database fields using LIKE operators.
* Search results displayed dynamically with minimal lag.

Challenges:

* Handling misspelled queries gracefully.
* Ensuring searches remained fast even as data size grew.

### **4.3.3 Symptom Consultation Module**

Users input symptoms and receive recommended formulations.

Implementation Highlights:

* Backend query joined Symptoms and Formulations tables via matching fields.
* Consultation results were ranked and displayed by relevance.

Challenges:

* Mapping multiple symptoms into one query.
* Preventing irrelevant results for vague symptoms.

### **4.3.4 Admin Panel Module**

Restricted dashboard for database management.

Implementation Highlights:

* Admin login protected via JWT role verification.
* Admin can add new drugs, edit existing details, or delete outdated entries.
* Frontend ensured confirmation prompts before destructive actions (e.g., deleting a drug).

Challenges:

* Designing UI simple enough for non-technical admins while preventing accidental data loss.

### **4.3.5 API Communication Module**

Manages consistent data exchange between frontend and backend.

Implementation Highlights:

* All APIs used JSON payloads.
* Standardized success and error response structures.
* Middleware handled token authentication automatically.

Challenges:

* Ensuring API versioning flexibility for future feature additions.

## **4.4 Key Implementation Decisions**

Many small yet critical decisions improved project maintainability and user experience.

### **4.4.1 Token Expiry for Sessions**

JWT tokens expired after 1 hour of inactivity to prevent session hijacking.

### **4.4.2 Input Validation at Both Frontend and Backend**

Frontend prevented invalid inputs early (like missing required fields), but backend double-checked all incoming data.

### **4.4.3 Lazy Loading in Vue.js**

Heavy components like Admin Dashboard were loaded only when needed, improving initial page load speeds.

### **4.4.4 Standardized API Responses**

Whether successful or failed, every API returned:

json

CopyEdit

{

"success": true,

"message": "Drug added successfully",

"data": {...}

}

or

json

CopyEdit

{

"success": false,

"message": "Invalid credentials",

"error": {...}

}

This standardization reduced frontend handling complexity.

### **4.4.5 Frontend Responsiveness**

CSS Flexbox and Grid layouts ensured that users experienced beautiful, clean UIs across desktops, tablets, and smartphones.

## **4.5 Sample Screenshots and UX Highlights**

Screenshots visually validated that system functionality was matched by user-friendliness:

### **4.5.1 Homepage**

* Welcoming banner introducing Unani Medicine exploration.
* Search bar prominently displayed.
* Quick access to Login/Register options.

### **4.5.2 Login and Registration Pages**

* Minimalistic forms with real-time validation.
* Helpful tooltips guiding user inputs.

### **4.5.3 Drug Search Results**

* Cards layout presenting each drug’s summary.
* Clicking a card revealed detailed descriptions.

**4.5.4 Symptom Consultation Page**

* Symptom input form with examples.
* Consultation results displayed with preparation instructions.

### **4.5.5 Admin Dashboard**

* Add/Edit/Delete options clearly separated.
* Confirmation popups before irreversible actions.

## **4.6 Reflections on Implementation**

Implementation taught lessons beyond technical skills:

* **Consistency Over Speed**: Rushing to code features caused bugs. Slow, methodical work avoided major rewrites.
* **Test-Driven Development Works**: Writing tests before complete features often caught mistakes early.
* **Document As You Build**: Regularly updating internal documentation reduced confusion when revisiting modules.
* **Frontend-Backend Communication Must Be Clear**: Tiny mismatches in expectations (field names, token handling) created big bugs.
* **User Perspective Matters Most**: No matter how technically “perfect” a feature was, it had to be usable and intuitive to non-developers.

Ultimately, implementation was an exercise in both technical craftsmanship and user empathy — building not just software, but an experience.

# **Chapter 5: Testing**

## **5.1 Introduction**

Testing constitutes the vital core that supports software projects to reach their successful goals. Tests were not a final step during development for the Knowledge-Based System (KBS) for Unani Medicine Formulations because testing ran through all coding decisions as an ongoing process.

The system aimed to manage genuine healthcare information so stability along with security and accurate performance became absolute priorities.  
 One unnoticed technical flaw poses a threat to user trust while holding the potential to supply faulty consultation recommendations.

The testing strategy alongside various **testing types** and **test case designs** **and bug tracking methods along** with used tools and system development insights through continuous feedback and improvements are presented in this chapter in detail.

## **5.2 Testing Strategy**

A strong testing strategy was designed from the outset based on three main pillars:

* **Prevention** (finding and fixing bugs early)
* **Detection** (discovering hidden flaws in integrated modules)
* **Correction** (learning from each error and improving overall system robustness)

We adopted an **incremental testing approach**, meaning that every new module, API, or frontend component was first tested in isolation (unit testing), then tested with others (integration testing), and finally tested as part of the full system (system testing).

Throughout the development cycle, manual and semi-automated testing practices were adopted using Postman for backend APIs and manual browser testing for frontend workflows.

A critical addition was **negative testing** — not only verifying that the system worked as expected but also trying to break it with wrong inputs, invalid API requests, and abnormal user behavior.

## **5.3 Types of Testing Conducted**

### **5.3.1 Unit Testing**

**Unit testing** was the first line of defense against bugs.

Each Vue.js component, Node.js API, and SQL query underwent isolated tests.  
 The goal was to make sure that each small piece of the system worked exactly as expected, without being affected by outside factors.

**Examples of unit tests:**

* **Frontend**:  
  + Input fields validated email formats before sending to backend.
  + Password field strength was enforced via regular expressions.
  + Vue components for login forms displayed instant error messages when empty fields were submitted.
* **Backend**:  
  + API /auth/register validated all required fields before creating a user.
  + Password hashing with bcrypt was unit tested to confirm that no passwords were stored in plaintext.
  + APIs rejected malformed JSON payloads gracefully.
* **Database**:  
  + SQL queries were tested for proper search results using hardcoded test entries.
  + Stored procedures returned correct drug and symptom matches.

We found that investing heavily in unit tests saved countless hours later during full system testing.  
 A bug detected early was often a bug cheaply fixed.

### **5.3.2 Integration Testing**

After units were verified individually, the focus shifted to **integration testing** — ensuring that different modules worked together smoothly.

**Examples of integration testing:**

* Registering a user via frontend and confirming that backend API created the corresponding record in SQL Server.
* Submitting symptoms via frontend consultation form and receiving formulation results through backend APIs.
* Ensuring that JWT tokens generated during login were correctly used in subsequent requests.

**Real problems encountered during integration testing:**

* **CORS errors**:  
   The browser blocked API requests because the server wasn't properly configured with CORS headers.  
   (Fixed by adding CORS middleware in Express.js.)
* **Field Mismatches**:  
   In some cases, frontend submitted "userEmail" but backend expected "email" — causing registration failures.  
   (Fixed by standardizing request payloads.)

Integration testing revealed that even tiny inconsistencies could cause entire workflows to fail — and taught the team to stay rigorous.

### **5.3.3 System Testing**

When individual parts passed unit and integration tests, the **full system** was tested end-to-end.

We acted like normal users — registering, logging in, searching for drugs, consulting for symptoms, managing entries as admins — without thinking about code or internals.

System Testing Scenarios:

* **Scenario 1**: New user registration → login → search for "Zanjabeel" → view drug details → logout.
* **Scenario 2**: Admin login → add new drug "Asrol" → edit "Zanjabeel" description → delete "OldEntry" drug.
* **Scenario 3**: Attempt SQL injection in symptom input → verify that system sanitizes inputs and blocks attacks.

System testing provided emotional highs and lows — it was satisfying to see complex workflows succeed, but frustrating when seemingly "working" code broke in unexpected ways during real usage.

### **5.3.4 User Acceptance Testing (UAT)**

Finally, **User Acceptance Testing** (UAT) provided an external reality check.

We recruited four testers with varied backgrounds:

* Two students of traditional medicine
* One IT student
* One general non-technical user (health enthusiast)

Each tester was asked to:

* Register an account.
* Search for familiar and unfamiliar drugs.
* Input symptoms like "cold," "headache," and find consultations.
* Reset their password.
* Act as "admins" and try unauthorized access to test security.

**Findings:**

* Testers appreciated simple layouts but asked for larger font sizes on mobile.
* Error messages needed to be clearer (e.g., instead of "Token missing", say "Please log in first").
* Non-technical users preferred seeing example symptoms on the consultation page.

Their feedback directly led to final round UI/UX tweaks before deployment.

## **5.4 Expanded Sample Test Cases**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Case ID** | **Description** | **Inputs** | **Expected Result** | **Actual Result** | **Status** |
| TC001 | Register with valid email/password | email: valid@test.com, password: Pass123! | Redirect to login | Success | ✅ |
| TC002 | Login with wrong password | email: valid@test.com, wrong password | Error: "Invalid Credentials" | Correct error | ✅ |
| TC003 | Search for known drug | "Zanjabeel" | Display drug details | Details shown | ✅ |
| TC004 | Search with empty field | (blank input) | Error: "Enter search keyword" | Correct | ✅ |
| TC005 | Consult with known symptom | "Fever" | Show formulations | Success | ✅ |
| TC006 | Consult with random text | "abcd1234" | No results, polite message | Success | ✅ |
| TC007 | Admin add new drug | Name: Asrol | Drug saved | Saved successfully | ✅ |
| TC008 | Admin edit drug details | Change description | Update confirmed | Edited successfully | ✅ |
| TC009 | Admin delete drug | Select entry | Entry deleted | Deleted successfully | ✅ |
| TC010 | Password reset request | Email: user@test.com | Reset link sent | Email sent successfully | ✅ |
| TC011 | Test SQL Injection attempt | Input malicious code | Blocked safely | Blocked | ✅ |
| TC012 | Mobile browsing experience | Different screen sizes | Responsive layout | Responsive layout verified | ✅ |
| TC013 | Refresh page after login | Active session maintained | Session remains active | Session maintained | ✅ |
| TC014 | Submit form with invalid email | "abc@" | Show error message | Error displayed correctly | ✅ |
| TC015 | High number of simultaneous requests | Simulated 50 API calls/sec | No server crash | Server stable | ✅ |

Table 2

✅ 15+ realistic, varied cases executed with full tracking!

## **5.5 Bug Tracker and Issues Log**

Bug tracking was detailed and ongoing.

|  |  |  |  |
| --- | --- | --- | --- |
| **Bug ID** | **Issue Description** | **Severity** | **Developer Notes** |
| B001 | Registration failed if password < 6 chars | Major | Added stricter frontend validation |
| B002 | Admin panel unauthorized access allowed | Critical | Secured with JWT middleware |
| B003 | Search bar allowed XSS scripts | Critical | Escaped user inputs |
| B004 | Consultation results UI overlapping text | Minor | Fixed CSS grid |
| B005 | Password reset email not formatted properly | Minor | Improved email template |
| B006 | Symptoms with special characters not returning results | Major | Improved backend parsing |
| B007 | CORS blocking API from frontend | Critical | Corrected server CORS config |
| B008 | SQL queries slowing under heavy load | Major | Added indexes on symptom search columns |

Table 3

The habit of immediate bug tracking, diagnosis, and resolution helped make the system very stable by the final phase.

## **5.6 Testing Tools and Techniques Used**

* **Postman**:  
   For backend API tests with chained request simulations (Login → Add Drug → Verify).
* **VS Code Debugger**:  
   Deep-dive into backend errors during server crashes.
* **SQL Query Analyzer**:  
   Used to optimize drug search queries, ensuring <1s query time under 10,000 records.
* **Manual Mobile Testing**:  
   Verified responsiveness using real smartphones, not just simulators.
* **Console Logging and Custom Error Pages**:  
   Improved frontend visibility when APIs failed (e.g., server down messages).

Testing wasn’t glamorous — but it was the difference between a project that "kind of works" and a system that "works reliably and is user-ready."

## **5.7 Expanded Reflections on Testing**

Testing felt like a second battle after coding.  
 Just when you think you are "done," testing shows you 50 things you missed.

Personal reflections:

* **Every success felt earned**:  
   Seeing a user log in and consult formulations without error felt genuinely satisfying.
* **Failures taught more than successes**:  
   Each bug revealed flaws not just in code, but in assumptions and design thinking.
* **Real users don’t behave like developers expect**:  
   UAT taught that users misclick, misinterpret, and make mistakes we never imagined — and systems must handle all of it gracefully.

Testing transformed the project from a nice demo into a resilient, dependable platform ready for real-world exposure.

# **Chapter 6: Deployment and User Manual**

## **6.1 Introduction**

Deployment is often mistakenly viewed as the "final step" in software development, a simple process of moving code to a server. In reality, deployment is a complex and strategic phase that tests every decision made during the design and development process.

It exposes how well the system handles real environments, real devices, real users — and most importantly, real mistakes.

For the Knowledge-Based System (KBS) for Unani Medicine Formulations, deployment wasn’t just about "getting it online." It was about building a foundation for reliability, security, maintainability, and future scalability.

This chapter walks through the complete deployment journey: environment setup, backend and frontend deployment, database configuration, real-world testing, user instructions, maintenance guidelines, and reflections on what deployment taught us about system resilience.

## **6.2 Deployment Strategy**

Before touching any servers or building production files, a clear deployment strategy was essential.  
Key goals shaped the approach:

* **Simplicity**: Use straightforward methods suitable for academic submission and real-user access.
* **Reliability**: Ensure the system remains available with minimal downtime.
* **Security**: Protect user data (especially passwords) even during transmission and storage.
* **Future Scalability**: Lay groundwork for migrating to scalable cloud solutions later if needed.

**Strategic decisions included**:

* Deploying frontend and backend separately (modular architecture).
* Hosting on basic HTTP servers initially (can easily upgrade later).
* Using environment variables for sensitive configurations.
* Keeping database backups manual but frequent.

This careful planning ensured that deployment wasn’t an afterthought but an integral part of system design.

## **6.3 Backend Deployment (Node.js + Express.js)**

The backend forms the heart of the KBS system — handling user authentication, processing searches, serving consultations, and managing admin operations.  
Deploying it properly was critical.

### **6.3.1 Backend Preparation**

First, the backend was cleaned up for production:

* **Removed Development Logs**: No unnecessary console outputs cluttering server logs.
* **Optimized APIs**: Ensured all API endpoints were secure and efficient.
* **Environment Variables Setup**: Created a .env file storing sensitive info like database URLs, JWT secrets, SMTP configs.

Example .env structure:

DB\_HOST=localhost

DB\_USER=admin

DB\_PASS=supersecret

JWT\_SECRET=myjwtsecretkey

SMTP\_SERVER=smtp.test.com

* **Tested Locally**: Postman was used to simulate real-world API flows.

### **6.3.2 Backend Hosting**

Since this was an academic deployment with limited funding, backend hosting was done using a basic local HTTP server (future-ready for cloud).

Steps:

1. Installed Node.js runtime on the target server.
2. Cloned the backend code repository.
3. Installed necessary packages using npm install.
4. Set environment variables manually on server.
5. Started the server using:

npm run start

1. Verified server APIs using Postman and browser console.

To maintain security:

* Ports were restricted using firewall rules.
* CORS headers allowed only frontend domain to access backend APIs.

### **6.3.3 Challenges Faced During Backend Deployment**

**CORS Errors**:  
First attempt to connect frontend to backend triggered CORS policy errors. Solution: Configured Express middleware to allow specific origins.

**Environment Variable Mistakes**:  
Initial server crashes were caused by missing or misnamed environment variables. Lesson: Always double-check .env files carefully.

**Session Timeout Bugs**:  
Tokens expired too quickly during initial testing, disrupting sessions. Extended token validity for academic demo sessions.

## **6.4 Frontend Deployment (Vue.js)**

The frontend, built using Vue.js, was the face of the system — the first impression users received.

### **6.4.1 Frontend Preparation**

First, the development version was prepared for production:

* **Build Optimization**: Ran:

npm run build

This generated minified, optimized static files inside /dist folder.

* **Environment Variables**:
  + API URLs were updated to point to live backend, not localhost.
  + Used .env.production file in Vue.js.

Example:

VUE\_APP\_API\_BASE\_URL=https://backend.myunaniapp.com/api

* **Performance Tuning**:  
  Lazy loading was implemented for routes like Admin Dashboard to reduce initial page load time.

### **6.4.2 Frontend Hosting**

Hosting was done on a basic HTTP static server (like live-server or simple file hosting platforms).

Steps:

1. Copy /dist folder contents to the hosting server.
2. Configure server to redirect all unknown routes (important for Vue SPA behavior).
3. Link frontend domain to backend API.

Browsers were tested across:

* Chrome
* Firefox
* Microsoft Edge
* Android Devices
* iPhones

Responsiveness, routing, and API calls were verified.

### **6.4.3 Challenges Faced During Frontend Deployment**

**Route Refreshing Failures**:  
SPA-style routing caused 404 errors on refresh. Solution: Configured server to fallback to index.html.

**Hardcoded API URLs**:  
Forgot to replace localhost URLs in some API calls, causing failures.  
Lesson: Always use environment variables for all URLs, no hardcoding.

## **6.5 Database Deployment (SQL Server + SSMS)**

The database stored precious knowledge — drug details, symptoms, formulations, user accounts.  
Its stability was critical.

### **6.5.1 SQL Server Setup**

Steps:

* Installed SQL Server on a secure machine.
* Configured SSMS (SQL Server Management Studio) for easy GUI management.
* Created production-ready tables based on final ER diagrams.

Example Tables:

* Users
* Drugs
* Symptoms
* Formulations

Indexes were applied on important search fields:

* drug\_name
* symptom\_name
* scientific\_name

### **6.5.2 Data Migration**

Initial database population was done using SQL INSERT scripts.  
Later, an admin dashboard was available to add/edit/delete drugs and formulations easily.

Regular backups scheduled manually every week (future automation planned).

### **6.5.3 Challenges in Database Deployment**

**Connection Timeouts**:  
Backend server initially timed out when querying heavy search operations.  
Solution: Increased SQL server timeout settings slightly and optimized indexes.

**Case Sensitivity Issues**:  
Some searches failed due to case sensitivity.  
Solution: Standardized queries to be case-insensitive (LOWER() functions in SQL).

## **6.6 Post-Deployment Testing**

Deployment isn't complete until post-deployment validation is done.

Testing Steps:

* **Register** ➔ **Login** ➔ **Search Drug** ➔ **Consult Symptoms** ➔ **Admin Actions** ➔ **Logout**✔️ Completed without errors.
* Simulated API load:  
  ✔️ 100 concurrent requests processed without server crash.
* Mobile Testing:  
  ✔️ Frontend responsive on real iOS and Android devices.
* Error Handling:  
  ✔️ Friendly messages on API/server failures.

## **6.7 User Manual**

A system is useless if users don't know how to use it!  
Here’s a simple, human-readable manual.

### **6.7.1 Accessing the System**

1. Open Google Chrome, Firefox, or any modern browser.
2. Visit the provided URL (e.g., https://myunaniapp.com).

### **6.7.2 Registration**

* Click **Register**.
* Fill your email, password (at least 6 characters).
* Confirm your password.
* Submit.
* You’ll receive a welcome message and can now login!

### **6.7.3 Login**

* Enter your registered email and password.
* Hit **Login**.
* Upon success, you’ll see the homepage with search and consultation options.

Forgot your password?

* Click **Forgot Password**, and check your email for reset instructions.

### **6.7.4 Searching Drugs**

* Type drug name (e.g., "Zanjabeel") in search bar.
* Click **Search**.
* Detailed drug cards will appear.

### **6.7.5 Symptom Consultation**

* Navigate to **Consultation**.
* Enter symptoms (e.g., "headache, fever").
* Click **Consult**.
* System recommends formulations!

### **6.7.6 Admin Dashboard (Admins Only)**

* After admin login, a **Dashboard** link appears.
* Admins can:
  + **Add New Drug**
  + **Edit Existing Drug**
  + **Delete Outdated Entry**
* All actions protected by JWT verification.

### **6.7.7 Logging Out**

* Click **Logout** at top right.
* Securely end your session.

## **6.8 Maintenance Guidelines**

A system that works today must keep working tomorrow —  
maintenance policies ensure survival.

### **6.8.1 Backend Maintenance**

* Monitor server logs weekly.
* Update NPM packages monthly.
* Renew JWT secret keys bi-annually.

### **6.8.2 Frontend Maintenance**

* Update Vue.js dependencies.
* Compress images/assets if page load times increase.

### **6.8.3 Database Maintenance**

* Weekly manual backups (eventually automate).
* Monthly integrity checks (DBCC CHECKDB commands).
* Archive old data into cold storage after a year.

### **6.8.4 Security Maintenance**

* Patch server OS and Node.js regularly.
* Monitor API access logs for anomalies.
* Implement CAPTCHA and 2FA in future upgrades.

## **6.9 Reflections on Deployment**

Deployment revealed surprising truths:

* **Localhost is a liar**:  
  Things that "worked on localhost" failed dramatically on real servers.
* **Environment management is critical**:  
  Missing variables, wrong API URLs, unsafe defaults — all became real-world problems.
* **Real users are unpredictable**:  
  Users refreshed pages halfway, switched devices, logged in twice — and systems had to handle it all gracefully.
* **Testing must continue even after 'success'**:  
  A system is never "done" — users always find new edge cases.

# **Chapter 7: Results, Evaluation, and Conclusion**

# **7.1 Introduction**

The Knowledge-Based System (KBS) for Unani Medicine Formulations was never just another academic project.  
 From the start, it was envisioned as a bridge — a bridge between ancient medical traditions and modern digital users, a bridge between scholars, students, and health enthusiasts.

But no project can truly claim success until it is reflected upon honestly:

* What worked well?
* What broke along the way?
* What was learned that no textbook could have taught?
* And how could the future versions be even better?

This chapter captures the heartbeat of the project: the victories, the battles, the lessons, the regrets, and the vision for what's next.

# **7.2 Summary of Achievements**

Despite the rollercoaster ride that defines any real-world development project, the KBS platform achieved remarkable milestones.

## **7.2.1 Technical Achievements**

* **End-to-End Full Stack System**:  
   Built a complete frontend-backend-database system from scratch using modern technologies (Vue.js, Node.js, SQL Server).
* **Responsive User Interface**:  
   Created a clean, mobile-friendly frontend with Vue.js, ensuring users on desktops, tablets, and smartphones had smooth experiences.
* **Secure Authentication System**:  
   Implemented JWT-based login/logout flows, password encryption using bcrypt, and token-based access control — critical for user trust.
* **Robust API Architecture**:  
   Designed and tested secure RESTful APIs for login, search, consultation, and admin operations.
* **Data Structuring**:  
   Digitized Unani drug data, mapped symptoms to formulations, and created relational database schemas that were efficient and scalable.

## **7.2.2 Functional Achievements**

* **Powerful Search Capabilities**:  
   Users can search by drug names, scientific names, or medicinal properties.
* **Consultation System**:  
   Users can enter symptoms and receive recommended formulations — making ancient healing methods accessible in seconds.
* **Admin Dashboard**:  
   Admins can manage drugs, formulations, and users without touching code or databases — ensuring sustainability.
* **Error Handling and User Guidance**:  
   User-friendly error messages ensured that even mistakes or technical issues didn’t create frustration.

## **7.2.3 User-Centric Achievements**

* **Simplicity and Accessibility**:  
   The platform feels intuitive, even for users unfamiliar with complex websites.
* **Real User Testing**:  
   Incorporated feedback from real testers (students, health enthusiasts), leading to critical UI improvements before final deployment.
* **Fast Performance**:  
   Optimized database queries, lazy loading, and minimal payloads ensured that the site loaded quickly even on modest internet connections.

"From a vague idea in early meetings to a live, usable system in the hands of users — this project traveled a long way and proved that thoughtful design, rigorous testing, and user feedback could transform theory into real-world value."

# **7.3 Challenges Faced**

No path worth walking is free from obstacles.  
 While celebrating achievements, it's equally important to recognize — and learn from — the hurdles encountered.

## **7.3.1 Technical Challenges**

### **7.3.1.1 Backend and Frontend Synchronization**

Initially, minor differences between frontend requests and backend expectations caused frustrating bugs.  
 For example, frontend sent field emailId, backend expected email — login failed despite correct credentials.

**Lesson**: Create and follow strict API documentation. Small inconsistencies cause major breakdowns.

### **7.3.1.2 CORS Issues**

When frontend tried calling backend APIs across different origins, browsers blocked them.

**Lesson**: Always set CORS headers properly and early. In production, CORS failures can completely kill a working system.

### **7.3.1.3 SQL Query Performance**

As the database grew, symptom consultations became slow. Some queries took over 5 seconds!

**Lesson**: Database indexing and query optimization aren't optional. They are mandatory for user experience.

## **7.3.2 Management and Planning Challenges**

* Underestimated time needed for real testing.
* Midway feature creep — temptation to add "cool new features" (AI symptom detection) which risked project deadlines.
* Occasional communication gaps when working in a team — missed updates caused duplication of work.

## **7.3.3 Resource Limitations**

* No paid servers — hosted locally for academic demo.
* Limited real-world users for feedback (couldn't involve actual Unani practitioners yet).
* No advanced load balancers or CDN optimizations possible.

# **7.4 Lessons Learned**

Projects teach lessons that no classroom lecture can.  
 Some lessons were technical; others were about teamwork, psychology, and life.

## **7.4.1 Lesson: Testing Must Start Early**

Every developer says "I’ll test later," and regrets it.  
 Starting unit tests and API simulations from day one saved the KBS project from nasty last-minute surprises.

## **7.4.2 Lesson: Assume Nothing About Users**

Users will:

* Type things you never expected.
* Click buttons you forgot existed.
* Get confused by interfaces you thought were obvious.

Always test with real users.

## **7.4.3 Lesson: Simplicity Wins**

Every time a feature or design was simplified, users responded better.  
 The less thinking users had to do, the more they trusted and enjoyed the system.

## **7.4.4 Lesson: Small Errors Cause Big Problems**

A missing token, a wrong API path, a small database typo — all caused major bugs.  
 Attention to tiny details isn't optional in real development — it’s survival.

**7.4.5 Lesson: Deployment is a Different Skillset**

Coding for localhost is completely different from deploying for the world.  
 Deployment demands attention to server configs, environment variables, security patches, and scalability.

"If development is building a castle, deployment is making sure it doesn't collapse the first time a strong wind blows."

# **7.5 Future Enhancements**

No system is perfect — especially not in its first version.  
 Several exciting future upgrades are planned to expand KBS’s impact.

## **7.5.1 Move to Scalable Cloud Hosting**

* Migrate backend and database to platforms like AWS, Azure, or DigitalOcean.
* Use autoscaling to handle sudden spikes in users.
* Global CDNs to speed up site load worldwide.

## **7.5.2 AI-Powered Consultations**

* Use machine learning models to recommend formulations based not just on keywords but symptom patterns.
* Build a recommendation engine that learns from user feedback.

## **7.5.3 Mobile Application Development**

* Create Android and iOS apps.
* Offline access: allow users to download parts of the database for consultation without internet.

## **7.5.4 Multilingual Support**

* Translate UI into Urdu, Arabic, Hindi, and English.
* Allow users to search in their native language.

## **7.5.5 Practitioner Collaboration Platform**

* Build verified practitioner accounts.
* Allow Unani doctors to submit new formulations for peer review.
* Turn the KBS into a living, growing body of traditional medicine knowledge.

# **7.6 Final Reflections**

Looking back, the journey of building the Knowledge-Based System feels like crossing a river on stepping stones.  
 Some stones were steady. Some slipped. Some required leaps of faith.

But step by step, the system grew from an abstract idea into a working, breathing digital platform —  
 preserving ancient healing knowledge for modern users.

**Highlights that stood out personally:**

* Seeing the first successful search result returned was magical.
* Watching real users search, consult, and smile at the interface was priceless.
* Fixing a bug at 2 AM and finally passing the Postman test suite felt better than winning a game.
* Explaining the project to a non-technical friend and seeing them "get it" felt like success beyond technical specs.

"The project didn't just preserve Unani formulations. It preserved something inside the developers too — a belief that technology, when used right, can serve tradition, history, and humanity."

## **7.7 Conclusion**

The Knowledge-Based System for Unani Medicine Formulations achieved its primary objectives:  
 digitizing ancient medical knowledge, making it accessible to modern users, and laying the foundation for future expansion.

The technical victories, the testing scars, the deployment battles, and the deep reflections together shaped a project that wasn’t just an academic exercise —  
 it became a **living, usable, and impactful system**.

There is still much to be done.  
 There are always more features to build, better hosting options to deploy, smarter AI models to train.  
 But this Version 1.0 — built with passion, rigor, and respect for ancient wisdom — will stand as proof that tradition and technology can not only coexist but thrive together.

And that is a legacy worth building.

# **References**

1. Hifzul Kabir. Introduction to Ilmul advia. Shamsher Publisher and Distributors, 2002.
2. Hifzul Kabir Morakkabat (Unani Formulations). Shamsher publisher and distributors. UP, 36:81–95, 2003.
3. R Lakshmi Poorna, Moghul Mymoon, and A Hariharan. Preservation and protection of traditional knowledge–diverse documentation initiatives across the globe. Current Science, pages 1240–1246, 2014. URL <https://www.researchgate.net/publication/280564503_Preservation_and_Protection_of_Traditional_Knowledge_-_Diverse_Documentation_Initiatives_across_the_Globe>
4. World Health Organization et al. WHO traditional medicine strategy: 2014- 2023. World Health Organization, 2013.
5. Node.js Foundation. (n.d.). *Node.js*. Retrieved from https://nodejs.org/en/
6. Microsoft. (n.d.). *SQL Server Documentation*. Retrieved from https://learn.microsoft.com/en-us/sql/sql-server/
7. Postman. (n.d.). *Postman API Platform*. Retrieved from https://www.postman.com/
8. World Health Organization. (2013). *WHO Traditional Medicine Strategy: 2014-2023*. Retrieved from https://www.who.int/publications/i/item/9789241506090
9. Nasir, A., & Ahmad, M. (2017). *Unani Medicine: A Comprehensive Review on the Unani System of Medicine*. Journal of Traditional and Complementary Medicine, 7(4), 391-396. https://doi.org/10.1016/j.jtcme.2016.08.004
10. Jackson, P. (1999). *Introduction to Expert Systems* (3rd ed.). Addison Wesley.
11. Balakrishna, C., & A. Shrinivasa, K. (2010). *Knowledge Based Systems in Healthcare: A Review*. International Journal of Information Technology and Knowledge Management, 2(2), 365-368.
12. GitHub, Inc. (n.d.). *GitHub – Where the World Builds Software*. Retrieved from https://github.com/
13. Mozilla Developer Network (MDN). (n.d.). *HTTP - Hypertext Transfer Protocol*. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP