Prince Mohammad Bin Fahd University

College of Computer Science and Engineering

Information Technology Department

Senior Project Report

HealthCare Credential Verification Application

Spring Semester 2024/2025

Learning Outcome Assessment III

Project Advisor: Dr.Nazeeruddin Mohammad

Names & ID & majors

Lamya Bakhurji 202001976 - Information Technology

Norah Alsubaie 202001750 - Information Technology

Nourah Alghubari 202001373 - Information Technology

Zaynab Taha 201802184 - Information Technology

Date of submission: 6th May 2025

Abstract

This group project focuses on the design and implementation of a Credential Verification System tailored for healthcare professionals. The primary goal is to streamline the traditionally complex and manual process of verifying professional credentials, ensuring that only qualified individuals are approved to practice in healthcare settings.

The system enables healthcare professionals to register, submit their credentials, and track the status of their verification. Administrators and regulatory bodies can efficiently manage, evaluate, and approve or reject credential submissions through an intuitive dashboard.

Developed using Flutter for the frontend, Node.js for the backend, and MongoDB as the database, the platform emphasizes security, data integrity, and role-based access control. Features such as multi-factor authentication, secure document uploads, and audit trails ensure compliance with healthcare data standards.

Through thorough testing and iteration, the system proved to be both reliable and scalable, offering a secure and efficient solution to credential management. This project showcases our ability to collaborate effectively and develop a real-world application that addresses a critical need in the healthcare sector.

Acknowledgments

We would like to express our sincere appreciation to everyone who contributed to the successful development of this credential verification system for healthcare professionals.

First and foremost, we extend our heartfelt gratitude to our project advisor, Dr. Nazeeruddin Mohammad, for his continuous guidance, insightful feedback, and unwavering encouragement throughout each phase of the project. His expertise played a pivotal role in shaping our understanding and approach.

We are also thankful to Prince Mohammad Bin Fahd University and the Information Technology Department for equipping us with the academic foundation and resources necessary to carry out this project effectively.

Special thanks to the dedicated team at Prince Mohammad Bin Fahd University whose real-world insights into credentialing processes and system operations enabled us to align our work with current industry standards and requirements.

Lastly, we are deeply grateful to our families and peers for their unwavering support, patience, and motivation throughout this journey.

Table of Contents

List of Figures

List of Tables

Digital Credential Verification System for healthcare workers

Introduction

Credentialing Verification Organizations (CVOs) are vital in ensuring the safety and trustworthiness of healthcare professionals by verifying their qualifications and experience. Their role is essential for safeguarding patient safety and maintaining the integrity of healthcare institutions. However, traditional methods of credential verification—relying on manual checks, fragmented data, and paper-based systems—are inefficient, costly, and prone to errors and fraud.

The urgency to modernize the credential verification process has never been more apparent. With the increasing complexity of healthcare delivery, the rising mobility of healthcare workers, and the sophistication of credential fraud, it’s critical to adopt innovative solutions. Delays in credential verification not only hinder timely hiring but can also worsen workforce shortages, affecting patient care. Healthcare workers, often stretched thin in fast-paced environments, are burdened with an additional challenge: navigating the slow and cumbersome credential verification process. Meanwhile, patients may be unknowingly at risk due to fraudulent credentials slipping through the cracks. Furthermore, the administrative burden of traditional systems diverts valuable resources from core clinical activities, resulting in higher operational costs.

This project proposes a Digital Credential Verification System (DCVS) specifically designed for healthcare professionals. By leveraging technologies such as secure cloud storage, digital signatures, and blockchain, the system offers a reliable, automated platform for managing, verifying, and sharing credentials. This isn't just a technical upgrade; it's a necessary evolution to ensure that healthcare workers are equipped and trusted to provide the best care possible. This digital approach not only accelerates verification processes but also protects against tampering and fraud by establishing immutable records of credential authenticity.

The platform will be user-friendly, enabling healthcare workers to upload, manage, and share credentials easily while empowering employers and credentialing bodies to conduct quick and reliable verification. Features like QR code-based verification and audit trails will ensure accessibility while maintaining high standards of data security and privacy.

The project aims to redefine credential management in healthcare by:

- **Enhancing Efficiency:** Automating verification to reduce turnaround times and administrative workload.
- **Strengthening Security:** Implementing tamper-proof digital records to prevent fraud.
- **Improving Usability:** Designing an intuitive interface for all stakeholders.
- **Supporting Regulatory Compliance:** Aligning with legal and accreditation standards for data integrity and privacy.

1.1 Background of the Study

Credential verification in healthcare is crucial for ensuring that healthcare workers have the necessary qualifications and certifications to provide safe care. Traditional credential verification methods—such as manual checks, physical document submissions, and third-party communication—are inefficient, error-prone, and vulnerable to fraud, leading to significant risks to patient safety and institutional integrity.

Credential verification is essential for ensuring that healthcare workers that are hired have the requirements necessary for their positions before their admission on board. These requirements are certifications, entrance and exit exams, internships, and experience.

As healthcare systems grow in complexity, organizations struggle to keep up with the volume of credentials needing verification. Outdated technology, manual record-keeping, and extensive paper systems further hinder efficiency. These issues often lead to delays in hiring, which can have serious consequences in staffing-sensitive areas like hospitals and clinics. In one instance, a hospital faced delays of several weeks in credentialing a critical surgeon, resulting in postponed surgeries and putting pressure on the healthcare team.

Credential fraud, including falsified qualifications, has been a growing concern, jeopardizing patient safety and eroding trust between healthcare workers, employers, and patients. This has become a serious problem. For example, there have been instances of unqualified professionals entering healthcare roles by using false credentials, putting patients' lives at risk. To combat these issues, healthcare organizations need to transition to more reliable, transparent, and efficient solutions that ensure the authenticity of professionals' credentials.

This study aims to design and implement a centralized, secure digital system for quick and accurate credential verification. It will also empower healthcare professionals by enabling them to manage and share their credentials in a controlled, digital environment. By moving away from traditional methods, the system will reduce errors, improve compliance, and expedite the credentialing process, allowing organizations to focus more on their core mission of patient care.

Initially, this solution will be tested within a hospital setting to validate its functionality and effectiveness. We are excited to see how this solution can alleviate some of the burdens faced by healthcare teams on a daily basis. The anticipated outcomes include greater verification efficiency, enhanced security, and reduced risk of fraud—ultimately improving patient care and the overall healthcare system.

2. Requirement Specification

2.1 Functional Requirements

2.1.1 User Registration and Authentication:

- Healthcare workers must be able to create an account and register their credentials securely.
- Multi-factor authentication (MFA) for secure login.
- Users (healthcare workers) can manage their profiles and update credentials when needed.

2.1.2 Credential Verification:

- The system must verify the authenticity of healthcare workers' credentials through internal processes.
- Credentials should be stored securely and protected against tampering or unauthorized changes.
- The verification process should be efficient, with real-time updates on the verification status.

2.1.3 Credential Management:

- Healthcare workers must be able to upload and store their certificates, licenses, and other credentials securely.
- Users should be able to easily view, share, and manage their credentials.
- Hospitals should have access to the verified credentials of their staff in an organized, easy-to-use format.

2.1.4 Hospital Admin Access:

- Hospital administrators must have the ability to search, verify, and manage staff credentials within the system.
- Admins should be able to view the credential history and verification status of healthcare workers.

2.1.5 Fraud Prevention:

- The system must ensure that all records are verified and protected against unauthorized changes.
- Any updates to credentials should be logged with an audit trail for transparency.

2.1.6 Internal Use in Hospitals:

- The system must be designed to operate internally within hospitals, allowing for secure management of staff credentials.
- Hospital staff (administrators and HR) should be able to easily access the system and perform verification tasks without needing specialized technical knowledge.

2.1.7 User-Friendly Interface:

- The system must have an intuitive and easy-to-navigate interface for both healthcare workers and hospital administrators.
- The design should be responsive and work across various devices (desktop, tablet, mobile).

2.1.8 Data Security:

- All healthcare worker data and credentials must be securely encrypted and stored.
- Access to the system should be role-based, with appropriate permissions for healthcare workers and hospital staff.

2.1.9 Audit Logs and Reports:

- The system should maintain an audit log of all actions performed (credential uploads, verification requests, updates).
- Reports should be available for administrators to monitor credential status, verification activities, and detect fraud attempts.

2.1.10 Cost Reduction and Automation:

- The system should automate the credential verification process to reduce manual checks and paperwork.
- It should allow for bulk verification of multiple healthcare worker credentials to save time and resources.

2.2 Security Requirements

2.2.1 Data Encryption

- All sensitive data **at rest** must be encrypted using industry standards such as **AES-256**.
- All data **in transit** must be secured via **TLS (Transport Layer Security)** to protect against eavesdropping or tampering.

2.2.2 Authentication and Access Security Design

- **Token-Based Authentication:** Use **JWT (JSON Web Tokens)** with defined expiration times for secure, stateless session management.
- **Secure Token Storage:** Store authentication tokens in secure, isolated local storage on the client device to prevent cross-site access.
- **Password Handling:** Store passwords using strong hashing algorithms such as **bcrypt**. Never store passwords in plaintext.
- **Multi-Factor Authentication (MFA):** Require MFA for enhanced account protection during login.

2.2.3 Role-Based Access Control (RBAC)

- Implement RBAC to enforce user-level permissions:
    - **Healthcare Workers:** Can register, log in, and upload/view only their own credentials.
    - **Administrators:** Have broader permissions to view, verify, and manage uploaded credentials system-wide.
- Ensure unauthorized access to restricted data is blocked based on role.

2.2.4 Audit Logs

- Log all critical system activities such as:
    - Credential uploads
    - Verification actions
    - Login attempts

- Ensure audit logs are **tamper-proof**, stored securely, and reviewed regularly for anomalies.

2.2.5 Secure Credential Upload

- Limit accepted file formats to **PDF** and **JPEG** only.
- Enforce file size limits to prevent resource abuse.
- Scan all uploaded documents using malware detection tools before storage or processing.

2.2.6 Backup and Disaster Recovery

- Perform automated, **regular backups** of encrypted data.
- Ensure disaster recovery processes enable system restoration within **15 minutes** of a failure or outage.

2.2.7 API and Network Security

- Secure all APIs using **API authentication keys** or OAuth2 protocols.
- Restrict API access to authorized applications and users only.
- Deploy **firewalls**, **intrusion detection**, and **monitoring tools** to detect and block unauthorized network access.

2.2.8 Session Management

- Implement **token expiration** and **refresh mechanisms** for user sessions.
- Auto-logout users after a period of inactivity to reduce session hijacking risks.

2.2.9 Regulatory and Legal Compliance

- Ensure the system complies with healthcare and data privacy regulations such as:
    - **HIPAA** (Health Insurance Portability and Accountability Act)

    - **GDPR** (General Data Protection Regulation)

- Require **explicit user consent** before collecting or processing personal and credential-related information.

- Provide transparency on data usage and retention policies.

2.3 Non-Functional Requirements

2.3.1 Performance:

- The system should handle up to 1,000 credential verification requests simultaneously without significant delays.

- Response time for verification results should not exceed 3 seconds under normal system load.

2.3.2 Scalability:

- The system must be capable of supporting additional hospitals or healthcare facilities with minimal configuration.

- It should efficiently manage an increasing number of users and credential records as adoption grows.

2.3.4 Security:

- All sensitive data must be encrypted both in transit and at rest using strong encryption algorithms (e.g., AES-256, TLS).

- Multi-factor authentication (MFA) must be enforced for all user logins.

- Role-Based Access Control (RBAC) should be implemented to prevent unauthorized access to protected data.

2.3.5 Availability:

- The system must maintain 99.9% uptime to ensure consistent and reliable access.

- Backup and disaster recovery procedures must support data restoration within 15 minutes of any system failure.

2.3.6 Usability:

- The interface should be simple, intuitive, and user-friendly for both healthcare workers and hospital administrators.
- The system must support multiple languages, including Arabic and English.
- Comprehensive user documentation and training materials must be available to guide first-time users.

2.3.7 Maintainability:

- The system should support easy upgrades, patching, and maintenance without interrupting ongoing services.
- Code should be modular, well-structured, and documented to enable rapid development and future enhancements.

2.3.8 Compliance:

- The system must comply with applicable data protection regulations such as GDPR and HIPAA.
- Audit logs must be retained and securely stored to track user activity and ensure compliance.

2.3.9 Interoperability:

- The system should integrate seamlessly with existing hospital management systems (HMS) and external APIs when required.
- Standard data formats (e.g., JSON, XML) should be used for data exchange between platforms.

2.3.10 Data Integrity:

- All credential data must be protected against tampering or unauthorized changes.
- Any modifications should be traceable and recorded in audit logs to maintain data trustworthiness.

2.3.11 Accessibility:

- The system must be fully accessible across multiple platforms, including web browsers, smartphones, and tablets.
- The design should support responsive layouts and accessibility standards to ensure usability by all users.

2.4 Constraints

2.4.1 Technology Constraints:

- The system must be developed using the specified technology stack: Flutter (backend), API (frontend), MongoDB (database), and Node.js (server-side development).
- The system must be compatible with common web technologies and mobile platforms to ensure cross-platform access.

2.4.2 Budget Constraints:

- Development and implementation costs must remain within the allocated project budget.
- Open-source tools and frameworks should be prioritized to minimize licensing and operational costs.

2.4.3 Timeline Constraints:

- The entire project must be completed within the academic semester or senior project deadline.
- A fully functional prototype should be ready for review within 8–12 weeks from project initiation.

2.4.4 Infrastructure Constraints:

- The system should be deployable within the existing IT infrastructure of hospitals.
- It should support both cloud-based and on-premises deployment options, depending on hospital preferences and policies.

2.4.5 Legal and Ethical Constraints:

- All healthcare worker and patient data must be used solely for credential verification purposes.
- Explicit consent must be obtained from users before collecting, processing, or sharing their credentials.
- The system must comply with privacy regulations and ethical standards to ensure the confidentiality and security of user data.

2.4.6 Resource Constraints:

- The project team consists of a limited number of members, so tasks must be efficiently distributed and managed.
- The system must optimize server and computing resources to handle anticipated loads without requiring additional hardware.

2.5 Risk Assessments

2.5.1 Technical Risks:

a. Integration Issues
Risk: Difficulty integrating the system with existing hospital systems or third-party APIs.
Likelihood: Medium
Impact: High
Mitigation:

- Use standard APIs and protocols for integration.
- Maintain clear documentation for both internal and external interfaces.
- Involve hospital IT teams early in the integration process.
- Conduct thorough testing of integration points.

b. Scalability Issues
Risk: The system may struggle to handle a large number of users or verification requests simultaneously.
Likelihood: Medium
Impact: High
Mitigation:

- Design the system with scalability in mind using microservices, horizontal scaling, and load balancing.

- Use MongoDB’s sharding and replication features for efficient data distribution.
- Conduct performance and stress testing to identify and address bottlenecks.

c. Data Corruption or Loss
Risk: Data stored in the database could become corrupted or lost due to system crashes, human error, or hardware failure.
Likelihood: Low
Impact: Critical
Mitigation:

- Implement regular, automated backups of MongoDB data.
- Use redundant storage and failover mechanisms.
- Periodically test the backup recovery process to ensure data can be restored quickly.

2.5.2 Security risks:

a. Unauthorized Access
Risk: Malicious users or attackers may gain unauthorized access to sensitive data.
Likelihood: Medium
Impact: High
Mitigation:

- Implement strong authentication (MFA) and session management.
- Use role-based access control (RBAC) to restrict access.
- Regularly audit user permissions and monitor for suspicious activity.

b. Data Breaches
Risk: Sensitive healthcare worker data could be exposed due to vulnerabilities or poor security practices.
Likelihood: Medium

Impact: Critical
Mitigation:

- Encrypt data at rest and in transit using AES-256 and TLS.
- Regularly update and patch all software components.
- Conduct regular security audits and penetration testing.

c. Malware from Uploaded Files
Risk: Users may upload files containing malware or malicious scripts.
Likelihood: Low
Impact: High
Mitigation:

- Restrict file types (PDF, JPEG, PNG) and enforce file size limits.
- Automatically scan uploaded files using antivirus tools.
- Sanitize file metadata and store uploads in isolated storage.

2.5.3 User-Related Risks:

a. User Errors
Risk: Users may accidentally delete or upload incorrect credentials.
Likelihood: Medium
Impact: Medium
Mitigation:

- Add confirmation dialogs for critical actions (delete/edit).
- Allow limited undo/revert capabilities.
- Provide clear tooltips and documentation within the interface.

b. Low User Adoption
Risk: Users (especially hospital admins) may resist adopting the new system.
Likelihood: Medium

Impact: High
Mitigation:

- Ensure the UI is user-friendly and available in multiple languages.
- Offer training sessions, demos, and support materials.
- Collect feedback and iteratively improve based on user experience.

c. Misuse of Access Privileges
Risk: Authorized users may misuse their access, intentionally or unintentionally.
Likelihood: Low
Impact: High
Mitigation:

- Maintain detailed audit logs of all actions.
- Regularly review admin activity and enforce access boundaries.
- Provide ethical use guidelines and training.

2.6 Applicable Standards

2.6.1 Data Protection Standards:

- GDPR: Ensures the protection of healthcare worker data within the European Union.
- HIPAA: Ensures the protection of healthcare data in the United States.

2.6.2 Blockchain Standards:

- ISO/IEC 27001: Provides guidelines to secure data stored on the blockchain.
- Hyperledger Fabric: Follows best practices to secure data and smart contracts.

2.6.3 Interoperability Standards:

- HL7 & FHIR: Ensure compatibility with other healthcare systems for data exchange.

2.6.4 Cybersecurity Standards:

- ISO/IEC 27032: Protects data from cybersecurity threats.
- NIST: Provides strategies to manage security risks.

2.6.5 Accessibility Standards:

- WCAG: Ensures the system is easy to use for all, including people with disabilities.

2.6.6 Quality Standards:

- ISO/IEC 9126: Ensures the software is reliable and user-friendly.
- ISO/IEC 25010: Ensures the software is secure and easy to use.

2.7 Project Plan

2.7.1 Project Overview

Project Title: Credential Verification System for Healthcare Workers Powered by Blockchain
Project Managers: [Norah Alghobari, Zaynab Taha, Norah Alsubiei, Lamya Bakhurji]
Start Date : [1-10-2024]
End Date: [4-5-2025]
Duration: [Approximation, 12 weeks]
Goal: Create a safe, expandable, and easy-to-use system that utilizes blockchain technology for credential validation in healthcare institutions.

2.7.2 Project Phases and Timelines

Phase 1: Planning and Requirements Collection (Weeks 1–2)

Actions:

1. Hold stakeholder meetings to gather detailed system and user requirements.
2. Define project scope, key goals, constraints, and success metrics.
3. Document functional and non-functional requirements (e.g. performance, security, usability).
4. Finalize technology stack (Flutter, Node.js, MongoDB, AWS/Azure).
5. Create the project charter and share it with all team members.

Deliverables:

- Requirements Specification Document
- Approved Project Charter

Phase 2: System Design (Weeks 3–4)

Actions:

1. Design the overall architecture using the MVVM pattern for Flutter.
2. Develop a database schema for user and credential data in MongoDB.
3. Define role-based access control (RBAC) for healthcare workers and administrators.
4. Create UI/UX wireframes and mockups for mobile and web.
5. Identify and document necessary API endpoints and integration points with hospital systems.

Deliverables:

- System Architecture Diagram
- Database Schema and Access Control Design
- UI/UX Mockups and Prototypes

Phase 3: Development (Weeks 5–9)

Actions:

1. Develop backend services in Node.js, including authentication, credential management, and sharing.
2. Implement secure REST APIs to connect frontend with backend and external systems.
3. Build the frontend in Flutter with modules for login, credential upload, viewing, and admin verification.
4. Implement core features:
    - User registration and secure authentication (with JWT + password hashing)
    - Credential submission, storage, and admin review

    - Credential sharing via time-limited links/QR
    - Secure document access with file type detection and viewer support

5. Handle state management using Provider and implement error handling and feedback UI.

Deliverables:

- Functional Backend and Database
- Cross-platform UI (Flutter for web and mobile)
- API Documentation and Codebase

Phase 4: Evaluation (Weeks 10–11)

Actions:

1. Conduct unit testing for API endpoints and UI components.
2. Perform integration testing (frontend + backend + MongoDB).
3. Simulate usage with 1,000+ users to assess performance and scalability.
4. Ensure compliance with data protection standards (HIPAA, GDPR).
5. Conduct user acceptance testing (UAT) with hospital staff and refine based on feedback.

Deliverables:

- Test Reports and Performance Metrics
- Finalized Codebase After Bug Fixes and Feedback

Phase 5: Implementation and Training (Week 12)

Actions:

1. Deploy the system to the hospital's cloud infrastructure (e.g., AWS or Azure).
2. Create and deliver training sessions, quick-start guides, and user manuals.
3. Run a final system verification and ensure readiness.

4. Launch the system and provide immediate post-deployment monitoring.

Deliverables:

- Deployed, Functional System
- Training Materials and User Documentation
- Final Implementation Report

Phase 6: Ongoing Support and Maintenance (Post-Deployment)

Actions:

1. Continuously monitor system performance, availability, and logs.
2. Provide ongoing technical support and resolve bugs or issues.
3. Collect user feedback and prioritize enhancements.
4. Add new features and security updates in iterative releases.

Deliverables:

- Maintenance Logs and Support Records
- Feature Update Reports and Release Notes

2.7.3 Project Milestones

Milestones	***Planned Date***	***Deliverables***
Requirements Gathering Completed	***Week 2***	***Requirements Specification Document***

System Design Finalized	***Week 4***	***Architecture Diagrams, Prototypes***
Backend and Frontend Development Complete	***Week 6***	***Functional System with Core Features***
Core System Development	***Week 9***	***Working System with Core Features***
Testing Completed	***Week 11***	***Test Results and Updated Codebase***
System Deployed	***Week 12***	***Live Operational System***

Table 1. Milestones table

2.7.4 Resource Allocation

Team Roles and Responsibilities:

1. **Project Manager:** Oversee project activities, manage timelines, and ensure project goals are met.
2. **Backend Developer:** Develop server-side functionality and APIs for credential management.
3. **Frontend Developer:** Build the user interface for the mobile and web application using Flutter.
4. **Quality Assurance (QA) Expert:**Test the application to ensure functionality, usability, and security.

2.7.5 Risk Management Plan

Key Risks and Mitigation:

Risk	Likelihood	Impact	Mitigation Plan
Delays in Requirement Gathering	Medium	High	Allocate buffer time for stakeholder input; conduct early and frequent consultations
Integration with Legacy Systems	Medium	High	Perform early system analysis; ensure use of standard APIs and interoperability formats (e.g., HL7/FHIR)
Data Breaches	Low	Critical	Implement encryption, access controls, and conduct regular security audits
Resistance from Users	Medium	Medium	Provide comprehensive training and user-friendly design

Table 2. Key Risks and Mitigation

2.7.6 Budget

Item	Estimated Cost
AI and Development Tools	$5,000
Cloud/Infrastructure Costs	$3,000
Team Salaries	$25,000
Training and Documentation	$2,000
Contingency	$2,000
Total Estimated Budget	$37,000

Table 3. Budget

2.7.7 Tools and Technologies

Frontend Framework:
Flutter: Used to build a cross-platform, responsive, and modern user interface for both web and mobile users.
Backend Framework:

- **Node.js**: Utilized to develop a scalable and high-performance backend API to handle user requests, authentication, and credential management.

Database Management System:

- **MongoDB**: A NoSQL document-based database used for flexible and efficient storage of user data and uploaded credentials.

AI Components:

- **Python Libraries (e.g., scikit-learn, TensorFlow)**: Integrated for intelligent features such as document classification, automated validation checks, or anomaly detection in credential submissions.

Security Tools and Libraries:

- **bcrypt**: Used for secure password hashing.
- **JWT (JSON Web Tokens)**: For secure token-based authentication.

- **MFA Libraries**: Used to implement multi-factor authentication.
- **File Scanners**: Employed to detect malware in uploaded credentials.

Version Control & Collaboration:

- **GitHub**: Used for version control, collaborative development, and issue tracking.

Testing Tools:

- **Postman**: Used for testing API endpoints.
- **Jest / Mocha**: Employed for backend unit testing.
- **Flutter Test**: For frontend widget and integration testing.

Hosting & Deployment Platforms:

- **Firebase / AWS / Heroku**: Used to host and deploy frontend and backend services with reliability and scalability.

2.7.8 Criteria for Success

System Uptime:

- Achieve a minimum of **99.9% uptime** after deployment to ensure continuous access for healthcare professionals and administrators.

Verification Accuracy:

- Ensure that **at least 95% of credentials** are verified successfully and accurately during User Acceptance Testing (UAT), with AI features assisting in classification or fraud detection.

User Satisfaction:

- Collect feedback from key stakeholders (healthcare workers and administrators), aiming for a minimum of **90% satisfaction** regarding system usability, performance, and security.

Security & Compliance:

- The system must fully comply with healthcare data protection regulations such as **HIPAA** and **GDPR**, as demonstrated through internal security assessments.

AI Feature Reliability:

- AI components (if applied for credential screening or anomaly detection) must maintain a **minimum precision and recall of 90%** during pilot testing.

2.7.9 Key Methods and their Flow

Login Flow

1. User enters email/password
2. Validation performed on input fields
3. Login request sent to API with credentials
4. On success, JWT token received and stored
5. User profile data fetched and stored
6. Navigation to appropriate home screen based on role
7. Error handling for failed login attempts

Document Viewing Flow

1. User selects credential to view
2. App shows loading indicator
3. Authentication token retrieved from storage
4. File download request sent with auth headers
5. Downloaded file saved to temporary location

6. File type detected from content-type header

7. Appropriate viewer opened (PDF or Image)

8. User interacts with document (zoom, scroll)

9. File removed when viewer is closed

Credential Sharing Flow

1. User selects credential to share

2. Share request sent to API

3. Server generates unique share token with expiry

4. QR code generated from share URL

5. User can set expiration time and regenerate

6. Share options (copy link, share via system)

7. Expiry validation before displaying

Document Handling Design

- **Secure Downloading**: Authentication headers for document access
- **File Type Support**: PDF and image viewers
- **Temporary Storage**: Files downloaded to temporary storage
- **Graceful Error Handling**: Network and file-related errors
- **Viewer Selection**: Appropriate viewer based on file type

3. Project Design

3.1. Conceptual Design

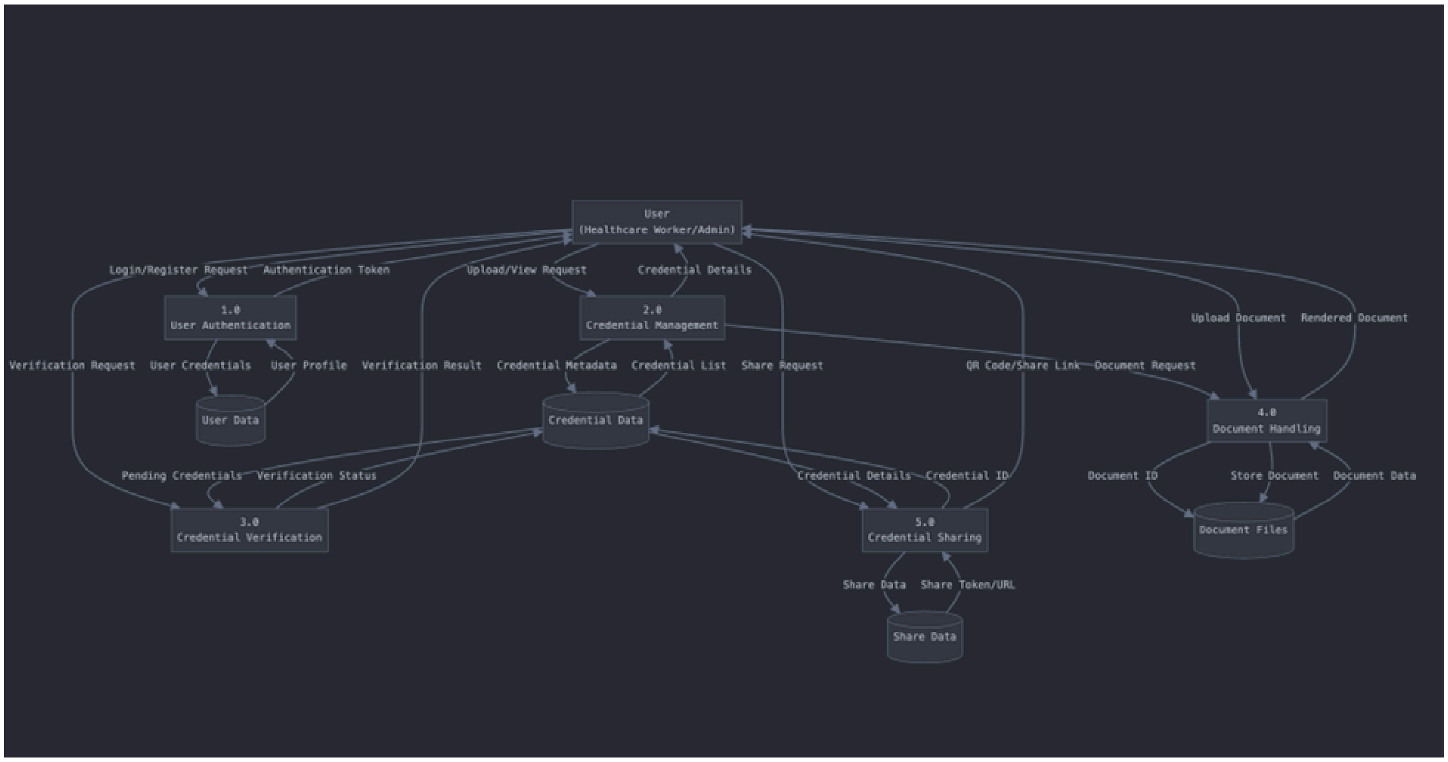
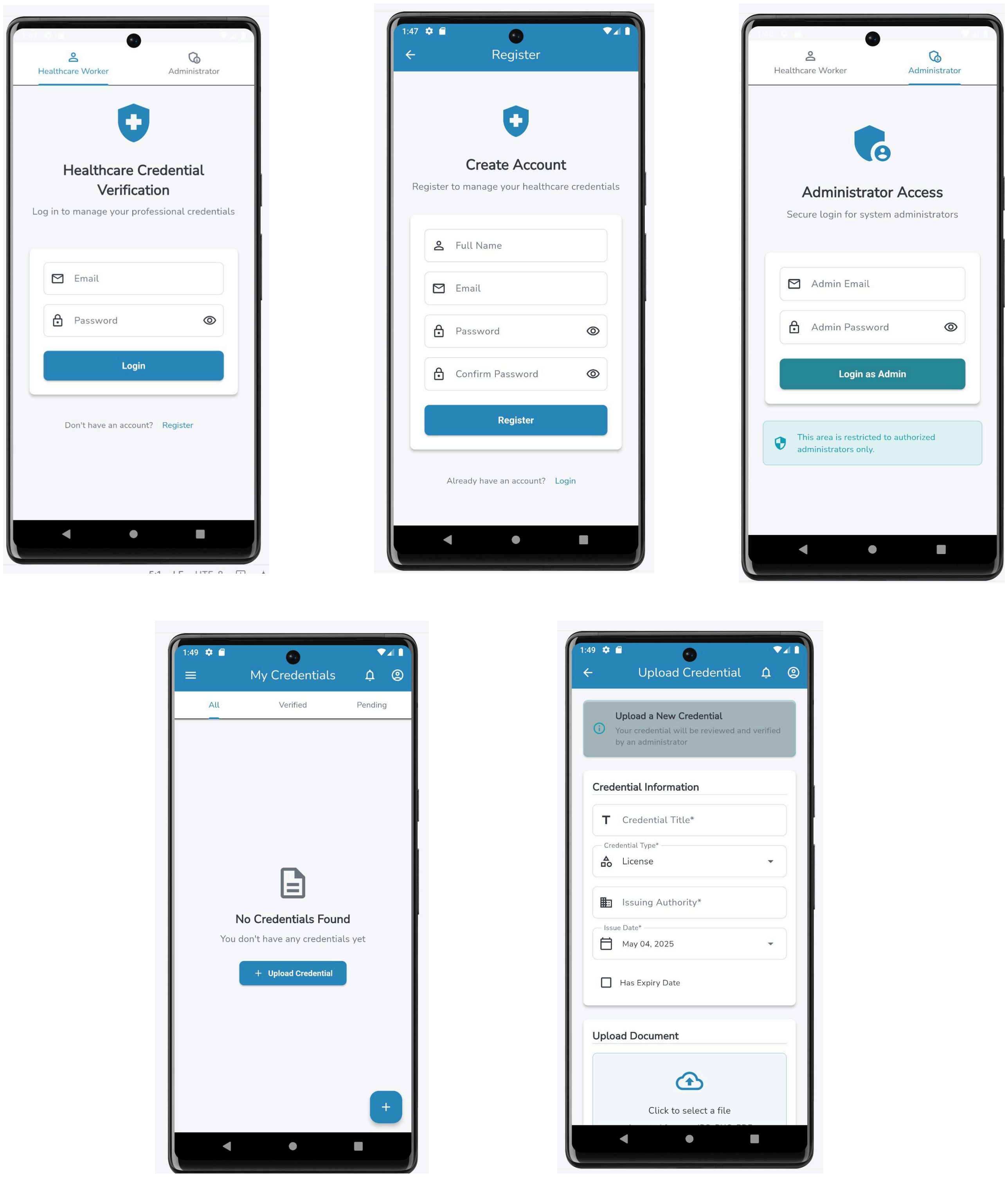


Figure 1. Conceptual Design – This figure illustrates the high-level architecture of the system, showcasing the key components and their relationships, including the user authentication, credential management, appointment scheduling, and audit logging processes.

The diagram illustrates the flow of interactions involved in managing user credentials within a digital credential verification system. It starts with the user initiating a login or registration request, which generates an authentication token for secure access. Once authenticated, the user can view their profile and access Credential Management, where they can manage their credentials. The process includes submitting a verification request, which checks the user's credentials and returns a verification result. Users can then share credential links or documents, such as uploading documents for verification. The system facilitates the sharing of credentials through a unique link or QR code, allowing others to access the shared documents securely. Throughout this flow, the system maintains a record of document handling, sharing status, and any pending verification statuses, ensuring a comprehensive and secure credential management process

3.2. User Interface Design



1:47

Register

Create Account

Register to manage your healthcare credentials

Full Name

Email

Password

Confirm Password

Register

Already have an account? Login

Healthcare Worker

Administrator

Healthcare Credential Verification

Log in to manage your professional credentials

Email

Password

Login

Don't have an account? Register

Healthcare Worker

Administrator

Administrator Access

Secure login for system administrators

Admin Email

Admin Password

Login as Admin

This area is restricted to authorized administrators only.

1:49

My Credentials

All

Verified

Pending

No Credentials Found

You don't have any credentials yet

+ Upload Credential

1:49

Upload Credential

Upload a New Credential

Your credential will be reviewed and verified by an administrator

Credential Information

Credential Title*

Credential Type*

License

Issuing Authority*

Issue Date*

May 04, 2025

Has Expiry Date

Upload Document

Click to select a file

Figure 2. User Interface Design for Credential Verification

Login Screens (Role-Based Access)

1. **Healthcare Worker Login**
    - **Purpose:** Allows licensed professionals to securely log in to manage and submit their credentials.
    - **Design Highlights:**
        - Simple email and password fields.
        - Role selection tab (Healthcare Worker / Administrator) on top for easy switching.
        - A Register link for new users to create an account.
        - Intuitive layout with branding and minimal distractions to maintain focus.
2. **Administrator Login**
    - **Purpose:** Provides secure access to authorized personnel (e.g., HR or Credentialing Officers).
    - **Design Highlights:**
        - Separate input fields for admin email and password for clarity.
        - Clearly marked section restricted to administrators only.
        - Maintains the same visual theme for consistency with the Healthcare Worker login page.

Register Screen

- **Purpose:** New healthcare workers can create an account to start uploading credentials.
- **Input Fields:**
    - Full Name
    - Email
    - Password & Confirm Password
- **Design Quality:**
    - Clean, user-friendly form layout with clear placeholders and icons for guidance.

    - Single call-to-action ("Register") to avoid confusion.
    - Responsive and mobile-friendly design to ensure accessibility on various devices.

Upload Credential Screen

- **Purpose:** Enables users to upload and submit their healthcare credentials for verification.
- **Input Sections:**
    - **Credential Title:** Free text input for the name/description of the credential.
    - **Credential Type:** Dropdown menu with options (e.g., License, Certificate).
    - **Issuing Authority:** Mandatory text input to specify the institution/organization.
    - **Issue Date:** Date picker component for selecting the issue date.
    - **Expiry Date (optional):** Checkbox for conditional field display if the credential has an expiration date.

- **Document Upload:**
    - Upload area clearly labeled with "Click to select a file" prompt.
    - Ensures that users can attach supporting documents.

- **Design Considerations:**
    - Logical grouping of fields to make the form intuitive and easy to navigate.
    - Guided inputs with mandatory fields to minimize user error.
    - Mobile-friendly design for a smooth experience across devices.

My Credentials Screen

- **Purpose:** Displays a user’s uploaded credentials and their verification status.
- **Tabs for Filtering:**

    - All, Verified, Pending – Allows users to quickly assess the status of their documents.
- **Empty State:**
    - Friendly message ("No Credentials Found") encourages users to upload.
    - Prominent "+ Upload Credential" button ensures ease of action.
- **Visual Feedback:**
    - Uses standard icons and proper alignment for better user comprehension.

3.3. Database Design

This credential verification system's database architecture is set up to ensure that we can safely access private data. Here, MongoDB, a NoSQL document-oriented database, is used by professionals and their credentials. The design places a strong emphasis on efficiency and flexibility while managing formal credential papers and dynamic user data. Three primary collections—User, Credential, and CredentialShare—form the basis of this approach. Each has a specific function in the verification process.

First, based on access control between administrative users and healthcare workers, the User collection holds crucial authorization and authentication data. The Credential collection contains comprehensive records of all licenses, certifications, and degrees that have been submitted, as well as information about their verification status and related documentation. Verified credentials can be securely shared thanks to the CredentialShare colle

User Collection

```
{
  _id: ObjectId,
  name: String,
  email: String,
  password: String (hashed),
  role: String (enum: "healthcare_worker", "admin"),
  profilePicture: String (optional URL),
  createdAt: DateTime,
  updatedAt: DateTime
}
```

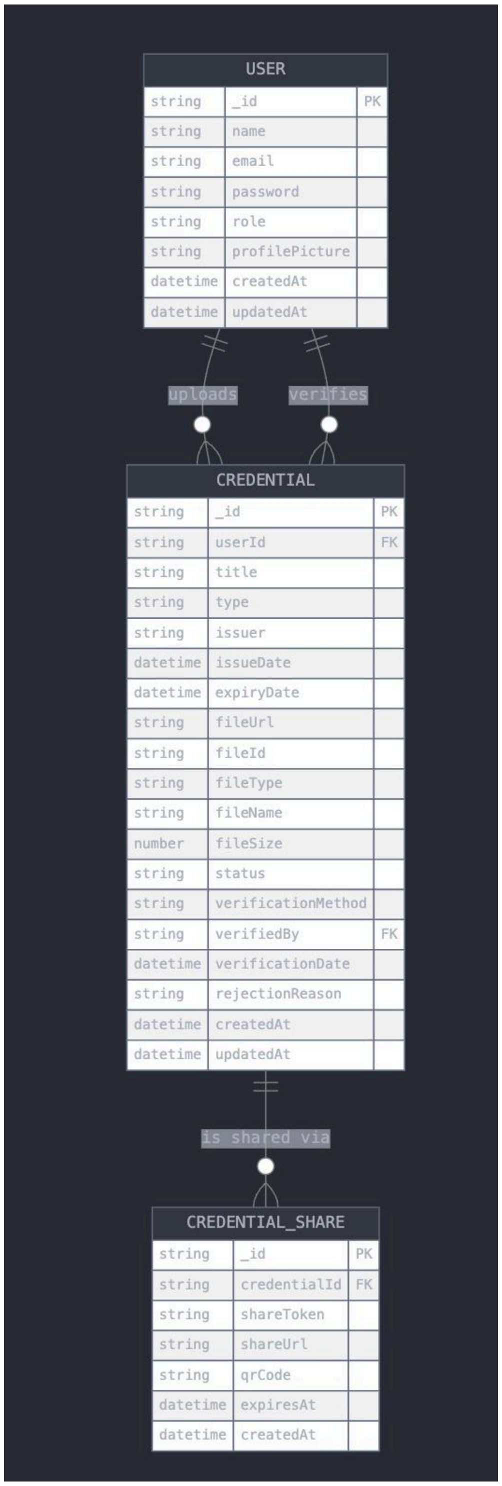
Credential Collection

```
{
  _id: ObjectId,
  userId: ObjectId (reference to User),
  title: String,
  type: String (enum: "license", "certification", "degree"),
  issuer: String,
  issueDate: DateTime,
  expiryDate: DateTime,
  fileUrl: String,
  fileId: String,
  fileType: String,
  fileName: String,
  fileSize: Number,
  status: String (enum: "pending", "verified", "rejected"),
  verificationMethod: String,
  verifiedBy: ObjectId (reference to admin User),
  verificationDate: DateTime,
  rejectionReason: String,
  createdAt: DateTime,
  updatedAt: DateTime
}
```

CredentialShare Collection

```
{
  _id: ObjectId,
  credentialId: ObjectId (reference to Credential),
  shareToken: String (unique),
  shareUrl: String,
  expiresAt: DateTime,
  createdAt: DateTime
}
```

3.4. ER Design



USER		
string	_id	PK
string	name	
string	email	
string	password	
string	role	
string	profilePicture	
datetime	createdAt	
datetime	updatedAt	

CREDENTIAL		
string	_id	PK
string	userId	FK
string	title	
string	type	
string	issuer	
datetime	issueDate	
datetime	expiryDate	
string	fileUrl	
string	fileId	
string	fileType	
string	fileName	
number	fileSize	
string	status	
string	verificationMethod	
string	verifiedBy	FK
datetime	verificationDate	
string	rejectionReason	
datetime	createdAt	
datetime	updatedAt	

CREDENTIAL_SHARE		
string	_id	PK
string	credentialId	FK
string	shareToken	
string	shareUrl	
string	qrCode	
datetime	expiresAt	
datetime	createdAt	

Figure 3. Entity-Relationship (ER) Diagram illustrating the structure and relationships between key entities in the credential verification system for healthcare professionals.

3.5. UML Design

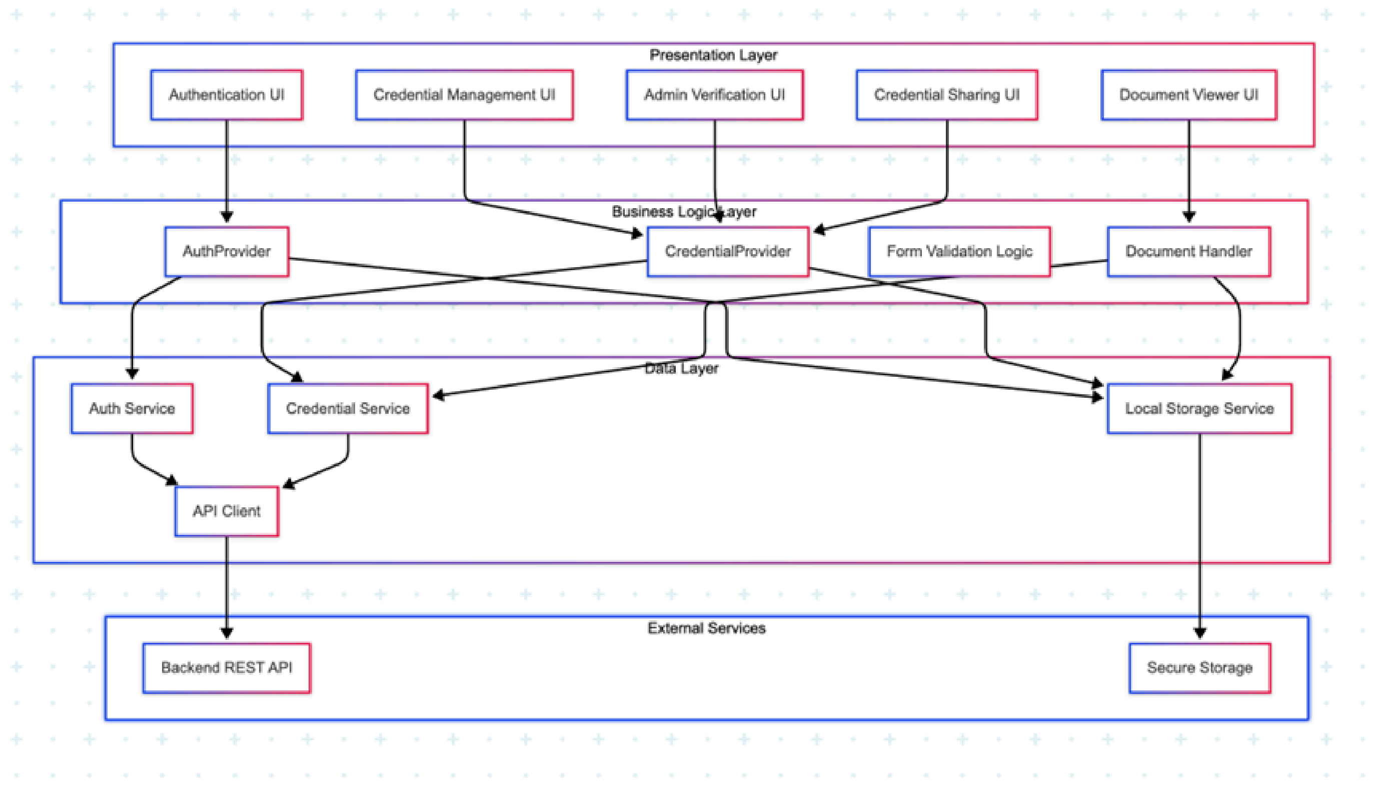


Figure 4. Unified Modeling Language (UML) diagram representing the system architecture, including use cases, classes, and interactions within the credential verification system.

The diagram presents a multi-layered architecture for a digital credential verification system, designed to maximize security and scalability by clearly separating system functions. At the top, the Presentation Layer offers various user interfaces for authentication, credential management, administrative verification, secure sharing, and document viewing. Beneath this, the Business Logic Layer manages core operations such as authentication, credential processing, form validation, and document handling. The Data Layer supports these functions with services for session management, credential storage and retrieval, local caching, and communication with external APIs. Finally, the architecture integrates with external services, including a backend REST API for processing requests and a secure storage solution for sensitive data. This structured approach ensures that each component operates independently yet cohesively, promoting maintainability, security, and efficient handling of digital credentials.

3.6. Software Design

3.6.1 Architecture Pattern: MVVM (Model-View-ViewModel)

- **Models**: Data structures representing entities (User, Credential)
- **Views**: Flutter UI widgets and screens
- **ViewModels**: Provider classes managing business logic and state

3.6.2 Authentication Flow

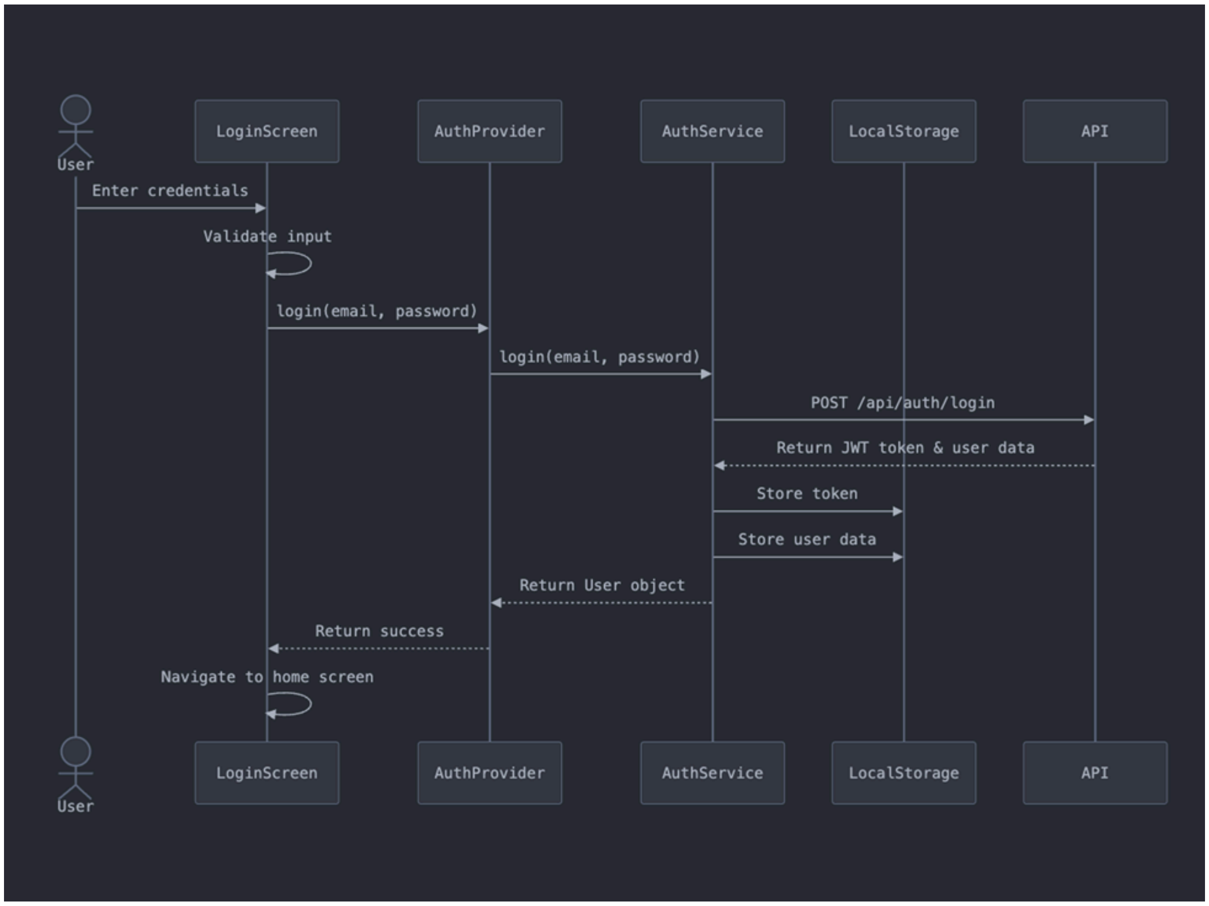


Figure 5. Authentication flow diagram demonstrating the process of user login, credential validation, and access control within the credential verification system.

The healthcare verification mechanism ensures that only authorized users, e.g., healthcare workers and admins, can access, upload, or verify credentials. The flow, the user authentication, and back-end perspectives. The authentication process starts with the end user, a healthcare worker or an admin, through the login interface. At this stage, the user types their email and password. Then the system validates the password to make sure the user is authorized. After the input is validated, the login credentials pass to the authentication provider. The service for authentication logic takes the task of verifying credentials to the authentication service. While the back end sends a POST request to the designated endpoint API, carrying the user's credentials. The API processes this request, cross-referencing the credentials with their hashed values in the MongoDB database, specifically in the User collection table. The server responds with a token along with associated user metadata such as role. After receiving the response, it stores the token and metadata in the browser's local storage. This allows the application to keep the user authenticated across different sessions and pages without repetitive logins. This also facilitates role-based access control to make sure healthcare workers are taken to the credential submission functionalities while admins are given access to verification tools. After successful storage of authentication data, the system returns the authenticated user object to the authentication provider and to the login screen to proceed. The user is then automatically redirected to their designated home screen according to their assigned role.

3.6.3 Credential Management Flow

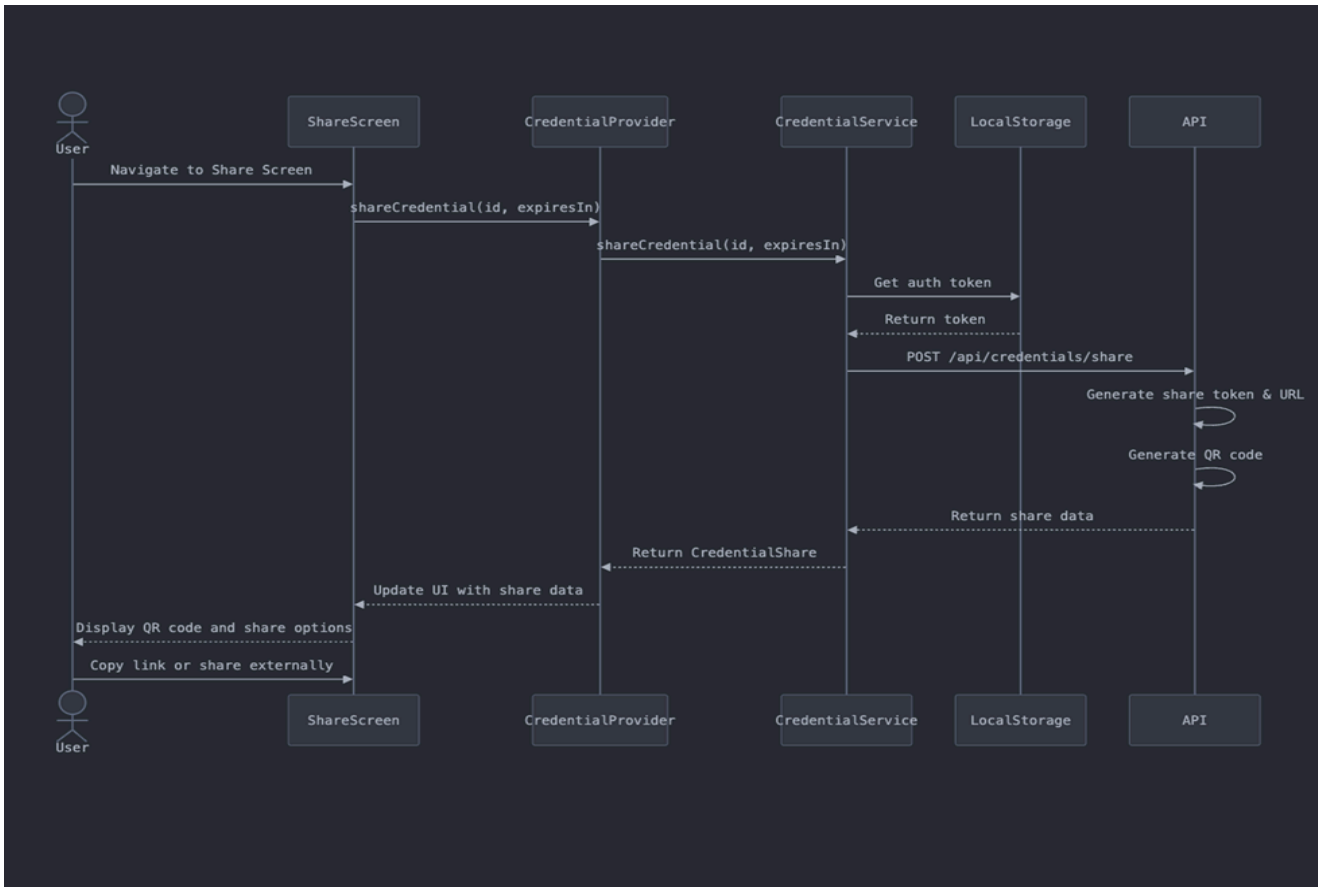


Figure 6. *Credential management flow diagram illustrating the process of uploading, verifying, updating, and securely storing healthcare professionals' credentials within the system.*

This diagram is a sequence diagram that illustrates the credential sharing process in a healthcare credential verification system. It visually represents how a healthcare worker can offload their verified credential using a QR code or a secure link through a series of interactions. The procedural flow for sharing the verified healthcare credentials begins with the user share screen and initiating a request to share credentials by calling the shareCredential(id, expiresIn) method, specifying the credential identifier and the expiration duration of the token. The request is then forwarded from the screen to the credential provider, which acts as a mediator between the UI and back end. The credential provider calls the shareCredential method on the credential service, which handles the business logic related to credential sharing. To access the credential, the service retrieves an authentication token from local storage, which has session data and identifies and authorizes the user's login. The token is retrieved and verified by the server. A POST request is made to the endpoint, which resides on the back end API layer. The endpoint generates the share token, share URL, and the QR code that can be used for external verification. After it successfully processes the request, it returns the generated share data to the credential service, which then passes this data back to the credential provider. Finally, the share screen is updated with the returned credential share information, including the tokenized link and the QR code. The user can view the QR code and other sharing options and can choose to copy the link or share it externally through social media applications. This does not compromise user privacy.

3.6.4 Credential Sharing Flow

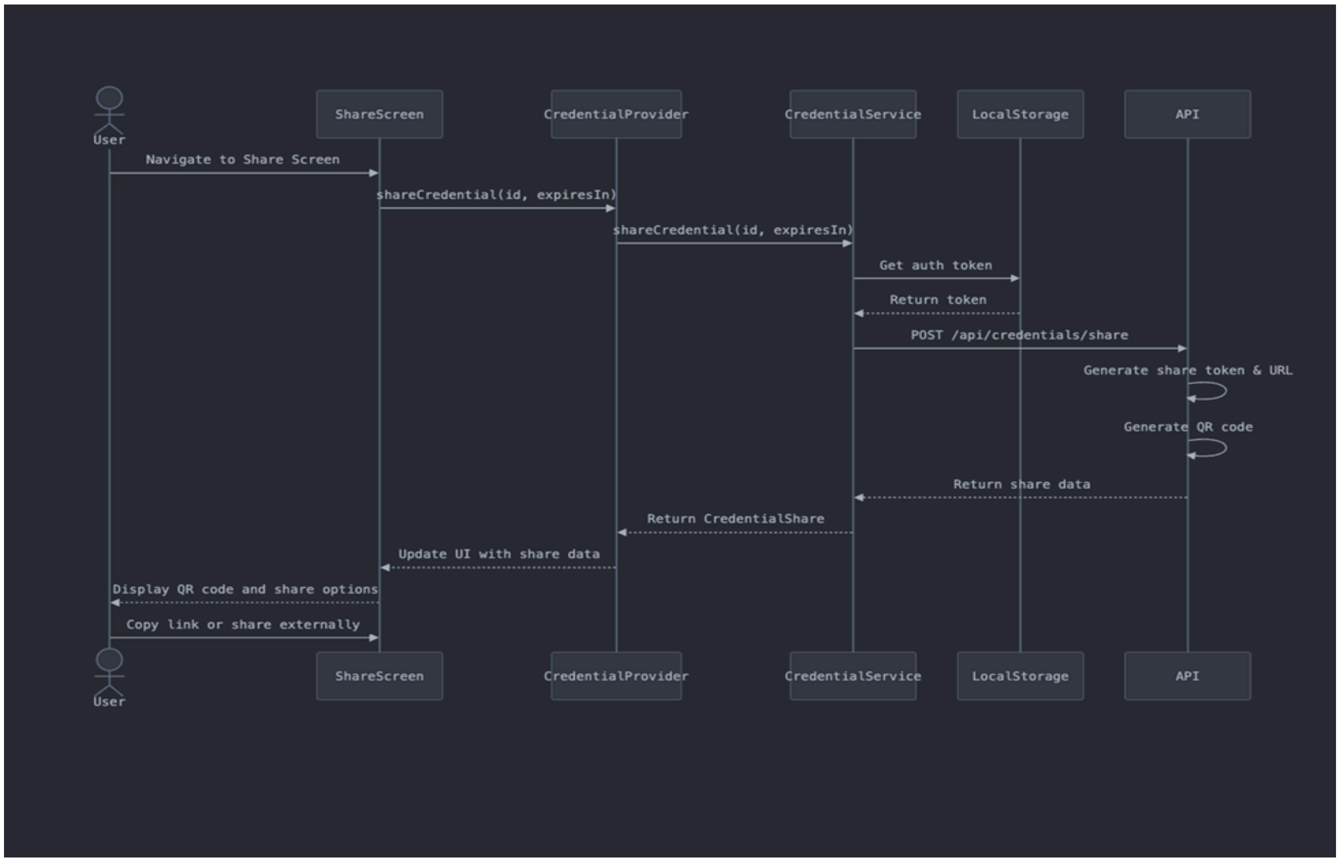


Figure 7. Credential sharing flow diagram depicting the secure process of granting access to verified credentials, including user consent, access permissions, and data protection measures.

The diagram depicts the Credential Sharing Flow, detailing how a user interacts with system components to securely share digital credentials. The process starts with the user accessing the Share Screen and selecting a credential by specifying its credentialId and expiration time (expiresIn). The CredentialProvider then requests a secure share token from the API, which is sent back to the user. Using this token, the user submits a POST request to the API endpoint (/api/credentials/share), prompting the system to generate a shareable link or QR code. The system responds with this share data, allowing the user to display or copy the link or code for distribution. This streamlined flow provides a secure and convenient way to share credentials across different communication platforms.

3.6.5 Module Structure

```
lib/
├── config/
│   ├── routes.dart          # App navigation routes
│   └── theme.dart           # UI themes and styles
├── core/
│   ├── models/
│   │   ├── user.dart        # User data model
│   │   ├── credential.dart # Credential data model
│   │   └── share_credential.dart # Share model
│   ├── providers/
│   │   ├── auth_provider.dart        # Authentication state
│   │   └── credential_provider.dart # Credential management
│   └── services/
│       ├── api_client.dart      # HTTP client
│       ├── auth_service.dart # Auth API calls
│       ├── credential_service.dart # Credential API calls
│       └── local_storage_service.dart # Local data storage
└── features/
        ├── auth/
```

```
│   ├── screens/
│   │   ├── login_screen.dart
│   │   └── register_screen.dart
│   └── widgets/
│     ├── login_form.dart
│     └── register_form.dart
      ├── credentials/
│   ├── screens/
│   │   ├── credentials_list_screen.dart
│   │   ├── credential_details_screen.dart
│   │   └── share_credential_screen.dart
│   └── widgets/
│     ├── credential_card.dart
│     └── document_viewer.dart
      ├── admin/
│   ├── screens/
│   │   └── pending_credentials_screen.dart
│   └── widgets/
│     └── verification_form.dart
      └── shared/
    └── widgets/
      ├── custom_app_bar.dart
      ├── loading_indicator.dart
      └── error_display.dart
```

The **features** directory is responsible for application-wide configurations and settings, housing two critical files: routes and theme. The routes file contains navigation routes and manages the different screens and their paths to ensure smooth transition between various parts of the app. The route here enables navigation, like login, credential, and admin panel. The theme handles the UI themes and styles, defining the app's color schemes, font styles, and other UI components. It ensures a consistent look across the entire app and allows for easy updating of the app's appearance.

The **core** directory contains the main logic of the application, housing models, providers, and services that handle data and business logic across different features. The **models** represent user, credential, and share credential. The **user** model defines the structure of the user object, with details like their name, email, and role. This model is used throughout the app whenever user data is needed. The **credential** model represents the credentials that users have, including ID, type, and validity dates. The **share credential** model contains data related to credentials shared with other users, such as expiry times and security tokens for shared access.

The **providers** folder contains the authentication provider and credential provider. The **auth_provider.dart** manages authentication states like login, logout, and session management. It plays a critical role in holding and updating the authentication status across the app, ensuring that the user session is consistent. The **credential_provider.dart** manages the state of user credentials, including fetching, creating, editing, and deleting credentials. It interacts with the data models and the UI to ensure the credentials are displayed properly.

The **services** folder contains three files: **api_client.dart**, **auth_service.dart**, **credential_service.dart**, and **local_storage_service.dart**. The **api_client.dart** is a critical component that handles all HTTP requests to external services, acting as the communication gateway between the app and the server to ensure that data is updated correctly. The **auth_service.dart** focuses on making API calls related to user authentication, such as login, token validation, and registration. The **credential_service.dart** handles operations related to credentials, like fetching the list of credentials or submitting new credentials to the server. The **local_storage_service.dart** manages data storage on the user's device and is responsible for securely storing sensitive data, like authentication tokens or user credentials, locally.

The **features** directory houses the primary functions of the app. It is divided into subcategories such as screens and widgets. The **auth** feature includes screens for user login and registration. The **login_screen.dart** provides fields for email and password and handles logic to trigger authentication. The **register_screen.dart** allows new users to sign up by providing their credentials and handles user creation and verification. The **widgets** directory contains reusable UI components, such as the **login_form.dart**, which gathers the user's credentials, and the **register_form.dart**, which gathers user information for registration.

The **credentials** feature contains screens that handle the management of credentials. The **credentials_list_screen.dart** displays a list of the user's credentials, fetched from the backend. Each credential can be tapped to view more details or share. The **credential_details_screen.dart** provides details about a specific credential. The **share_credential_screen.dart** allows users to securely share credentials with others. The **widgets** folder in this feature includes the **credential_card.dart**, which displays a summary of the user's credential in a card format, showing the type and basic metadata, and the **document_viewer.dart**, which enables users to view documents such as PDFs or images in a user-friendly manner.

The **admin** feature includes screens for administrators to manage user credentials. The **pending_credentials_screen.dart** displays credentials that are awaiting approval by an administrator. The **widgets** folder contains the **verification_form.dart**, which is used by admins to verify the credentials submitted by users. This form may include options to approve or reject credentials.

The **shared** directory houses reusable UI components across various features. The **custom_app_bar.dart** defines a consistent top UI for navigation, including the app title and any action buttons. The **loading_indicator.dart** is used to show that the app is processing data, and the **error_display.dart** is used to display error messages when something goes wrong.

3.6.6 Authentication Security Design

- **Token-based Authentication**: JWT tokens with expiration
- **Secure Storage**: Tokens stored in secure local storage
- **Role-based Access Control**: Different permissions for healthcare workers vs admins
- **Password Handling**: Hashed passwords, never stored in plain text
- **Session Management**: Token refresh and expiration handling

3.6.7 Document Handling Design

- **Secure Downloading**: Authentication headers for document access
- **File Type Support**: PDF and image viewers
- **Temporary Storage**: Files downloaded to temporary storage
- **Graceful Error Handling**: Network and file-related errors
- **Viewer Selection**: Appropriate viewer based on file type

3.6.8 Key Methods and Their Flow

Login Flow

1. User enters email/password

2. Validation performed on input fields

3. Login request sent to API with credentials

4. On success, JWT token received and stored

5. User profile data fetched and stored

6. Navigation to appropriate home screen based on role

7. Error handling for failed login attempts

Document Viewing Flow

1. User selects credential to view

2. App shows loading indicator

3. Authentication token retrieved from storage

4. File download request sent with auth headers

5. Downloaded file saved to temporary location

6. File type detected from content-type header

7. Appropriate viewer opened (PDF or Image)

8. User interacts with document (zoom, scroll)

9. File removed when viewer is closed

Credential Sharing Flow

1. User selects credential to share

2. Share request sent to API

3. Server generates unique share token with expiry

4. QR code generated from share URL

5. User can set expiration time and regenerate

6. Share options (copy link, share via system)

7. Expiry validation before displaying

3.7. Design Constraints

The development of the healthcare credentialing system was guided by several design constraints that influenced the architectural choices, technology stack, and implementation decisions. These constraints were considered to ensure regulatory compliance, system efficiency, and user satisfaction.

3.6.1 Data Privacy and Security Regulations

- Constraint: Compliance with healthcare data regulations such as Saudi Health Information Exchange Policies, HIPAA (if applicable), or similar standards.
- Impact: All patient and credential data must be encrypted, stored securely, and only accessible to authorized users. Multi-factor authentication (MFA) and audit trails were mandatory for user actions.

3.6.2 Role-Based Access Control (RBAC)

- Constraint: The system must strictly separate access privileges between different user types: Admins, Healthcare Professionals, Evaluation Committees, and Regulatory Authorities.
- Impact: The design required customized dashboards, backend route restrictions, and fine-grained permission management.

3.6.3 Technology Stack Limitation

- Constraint: Use of Flutter for frontend, Node.js for backend, and MongoDB as the primary database.
- Impact: All design decisions had to align with the capabilities of these technologies, such as asynchronous handling in Node.js and document-based schema design in MongoDB.

3.6.4 Offline Document Storage

- Constraint: Large documents (e.g., scanned certificates) should not be stored directly in the main database to avoid performance issues.
- Impact: External secure file storage services or cloud buckets were considered for storing large files, with references stored in MongoDB.

3.6.5 Performance and Scalability

- Constraint: The system should support concurrent access by multiple users without degradation in performance.

- Impact: API endpoints were optimized using pagination, indexing, and load management techniques to maintain responsiveness under load.

3.6.6 User Experience (UX) Requirements

- Constraint: The interface must be accessible, responsive, and usable for healthcare professionals with varying degrees of technical proficiency.
- Impact: A clean and intuitive design was implemented using Flutter's responsive layout system and guided user flows.

3.6.7 Auditability

- Constraint: Every credential-related action must be traceable.
- Impact: The system logs key user actions (e.g., login, credential submission, approval, rejection) with timestamps for compliance and transparency.

3.6.8 Language and Localization

- Constraint: The system may need to support both Arabic and English for broader accessibility within Saudi Arabia.
- Impact: The frontend was designed with internationalization (i18n) support in mind, enabling seamless switching between languages.

4. Implementation

This section describes the practical development of the healthcare credentialing system. It outlines the technologies used, the system's architecture, and the implementation of each core component.

4.1.1 System Architecture

The system is based on a three-tier architecture, consisting of:

- Frontend: Built with Flutter, providing a responsive and interactive user interface for patients, healthcare professionals, and administrators.
- Backend: Developed using Node.js and Express.js, which handles the business logic, routes, and API endpoints.
- Database: MongoDB is used as a NoSQL database for storing user data, credentials, appointments, and logs.

A secure API-based communication layer enables smooth interaction between the frontend and backend. Role-based access control ensures that users only access features appropriate to their role.

4.1.2 Frontend Development

The Flutter frontend is designed to be intuitive and accessible for all user roles.

Key features:

- Login Page: Includes email/government ID login, “Remember Me,” and “Forgot Password.”

```
// POST /api/login

const login = async (req, res) => {

 const { identifier, password, rememberMe } = req.body; // identifier = email or gov ID

 const user = await User.findOne({

    ☐    $or: [{ email: identifier }, { govID: identifier }],

 });

 if (!user || !(await bcrypt.compare(password, user.password))) {

  return res.status(401).json({ error: 'Invalid credentials' });

 }

 const accessToken = jwt.sign({ id: user._id }, process.env.ACCESS_TOKEN_SECRET, {

  expiresIn: rememberMe ? '7d' : '15m',

 });

 const refreshToken = jwt.sign({ id: user._id }, process.env.REFRESH_TOKEN_SECRET, {

  expiresIn: '7d',

 });

 user.refreshToken = { token: refreshToken };

 await user.save();

 res.json({ accessToken, refreshToken });

};
```

- Multi-factor Authentication (MFA): Users are required to verify their identity using a second factor after logging in.

```
// POST /api/mfa/send

const sendMfaCode = async (req, res) => {

  const { userId } = req.body;

  const code = Math.floor(100000 + Math.random() * 900000).toString();

  const expiry = Date.now() + 5 * 60 * 1000; // 5 minutes

  await User.findByIdAndUpdate(userId, {

    mfaCode: { code, expiry },

  });

  // Simulate sending via email/SMS

  console.log(`MFA code for user ${userId}: ${code}`);

  res.json({ message: 'MFA code sent' });

};

// POST /api/mfa/verify

const verifyMfaCode = async (req, res) => {

  const { userId, code } = req.body;

  const user = await User.findById(userId);

  if (!user || !user.mfaCode || user.mfaCode.code !== code || Date.now() > user.mfaCode.expiry) {

    return res.status(400).json({ error: 'Invalid or expired MFA code' });

  }

  res.json({ success: true, message: 'MFA verified' });

};
```

- Patient Dashboard: Displays appointment status, medical history, test results, and prescriptions.

```
// GET /api/patient/dashboard

const getPatientDashboard = async (req, res) => {
const userId = req.user.id;
 const patient = await Patient.findOne({ user: userId })
  .populate('appointments')
  .populate('medicalHistory')
  .populate('testResults')
  .populate('prescriptions');
 if (!patient) return res.status(404).json({ error: 'Patient not found' });
 res.json({
  appointments: patient.appointments,
  history: patient.medicalHistory,
  testResults: patient.testResults,
  prescriptions: patient.prescriptions,
 });
};
```

- Credential Submission Interface: For healthcare professionals to upload documents and fill credentialing forms.

```
// POST /api/credentials/submit
const submitCredentials = async (req, res) => {
  const { userId, documents, formData } = req.body;
  const credential = new Credential({
    user: userId,
    documents,
    formData,
    status: 'Pending',
  });
  await credential.save();
  res.json({ message: 'Credentials submitted successfully', credential });
};
```

- Notification Center: Shows alerts about appointment reminders, verification results, or missing documents.

```
// GET /api/notifications
const getNotifications = async (req, res) => {
  const userId = req.user.id;
  const notifications = await Notification.find({ user: userId }).sort({ createdAt: -1 });
  res.json({ notifications });
};
```

4.1.3 Backend Development

The backend was developed using Node.js with Express.js to manage the server-side logic.

Main functionalities:

- User Authentication: Handled using JWT (JSON Web Tokens).

```
// POST /api/login (User Login)
const login = async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });
  if (!user || !(await bcrypt.compare(password, user.password))) {
    return res.status(401).json({ error: 'Invalid credentials' });
  }
  const accessToken = jwt.sign({ id: user._id }, process.env.ACCESS_TOKEN_SECRET, { expiresIn: '1h' });
  const refreshToken = jwt.sign({ id: user._id }, process.env.REFRESH_TOKEN_SECRET, { expiresIn: '7d' });
  user.refreshToken = { token: refreshToken };
  await user.save();
```

```
 res.json({ accessToken, refreshToken });
};


// POST /api/refresh (Refresh Token)
const refreshToken = async (req, res) => {
 const { refreshToken } = req.body;
 if (!refreshToken) return res.status(401).json({ error: 'Refresh token missing' });
 try {
  const decoded = jwt.verify(refreshToken, process.env.REFRESH_TOKEN_SECRET);
  const newAccessToken = jwt.sign({ id: decoded.id }, process.env.ACCESS_TOKEN_SECRET, { expiresIn: '1h' });
  res.json({ accessToken: newAccessToken });
 } catch (err) {
  return res.status(403).json({ error: 'Invalid or expired refresh token' });
 }
};
```

- Credential Management: APIs to submit, update, and verify credentials.

```
// POST /api/credentials/submit (Submit Credential)
const submitCredential = async (req, res) => {
 const { userId, documents, credentialDetails } = req.body;
 const newCredential = new Credential({
  user: userId,
  documents,
  credentialDetails,
  status: 'Pending',
 });
```

```
  await newCredential.save();
  // Log the action in the Audit Log
  const auditLog = new AuditLog({
    action: 'Submit Credential',
    user: userId,
    details: `Credential submitted by user ${userId}`,
  });
  await auditLog.save();
  res.json({ message: 'Credential submitted successfully' });
};
```

- Admin Controls: Endpoints to approve, reject, or edit credential records.

```
// POST /api/credentials/approve/:id (Admin Approve Credential)
const approveCredential = async (req, res) => {
  const credentialId = req.params.id;
  const credential = await Credential.findById(credentialId);
  if (!credential) {
    return res.status(404).json({ error: 'Credential not found' });
  }
  credential.status = 'Approved';
  await credential.save();
  // Log the action in the Audit Log
  const auditLog = new AuditLog({
    action: 'Approve Credential',
    user: req.user.id,
    details: `Credential approved for credential ID ${credentialId}`,
  });
  await auditLog.save();
  res.json({ message: 'Credential approved successfully', credential });
};
// Reject Credential
const rejectCredential = async (req, res) => {
  const credentialId = req.params.id;
  const credential = await Credential.findById(credentialId);
  if (!credential) {
    return res.status(404).json({ error: 'Credential not found' });
  }
  credential.status = 'Rejected';
  await credential.save();
  // Log the action in the Audit Log
  const auditLog = new AuditLog({
    action: 'Reject Credential',
    user: req.user.id,
    details: `Credential rejected for credential ID ${credentialId}`,
  });
  await auditLog.save();
  res.json({ message: 'Credential rejected successfully', credential });
};
```

- Audit Logs: Every action (e.g., data update, approval) is recorded for tracking and compliance.

4.1.4 Database Integration

MongoDB was chosen for its flexibility in storing diverse and dynamic healthcare data. Collections include:

- Users: Stores login credentials, roles, and personal details.
- Credentials: Contains uploaded credential documents, status (pending, approved, rejected), and reviewer notes.
- Appointments: Records patient bookings, status, and doctor assignment.
- Audit Logs: Maintains a trail of all activities for compliance tracking.

Each document is associated with a unique identifier and timestamps for traceability.

4.1.5 Security Measures

Security was a key concern due to the sensitivity of patient and credential data. Measures implemented include:

- Multi-factor Authentication (MFA): Adds an extra layer of protection during login.
- Encryption: Sensitive data such as passwords and documents, are encrypted using industry-standard algorithms.
- Role-Based Access Control (RBAC): Only authorized users can access specific functions.
- Input Validation: Prevents common vulnerabilities like SQL injection and cross-site scripting (XSS).

4.1.6 Admin Panel

An admin-specific interface was developed to support the following functionalities:

- Search, view, and update credentialing records.
- Add notes or request resubmission of missing documents.
- Monitor pending verifications and approve or reject applications.
- Generate reports on credentialing activity over specific periods.

5. Testing and Verification

Testing and verification were essential in ensuring the healthcare credentialing system's reliability, security, and usability. Multiple types of testing were performed across both frontend and backend components to validate system performance under various scenarios.

5.1.1 Test Plan

The testing process focused on verifying the functionality, security, and user experience of the system’s key components:

- User authentication (including MFA)
- Role-based access and permissions
- Credential submission and review workflow
- Appointment scheduling
- Notification system
- Admin dashboard operations

5.1.2 Types of Testing Performed

5.2.1 Unit testing: Tested individual functions such as input validation, authentication logic, and database queries.

5.2.1 Integration Testing: Verified seamless interaction between frontend and backend via API endpoints.

5.2.3 System Testing: Conducted end-to-end tests to simulate real-world user interactions from login to credential verification.

5.2.4 Security Testing: Checked login vulnerabilities, data encryption, access restrictions, and token-based authentication.

5.2.5 Usability Testing: Collected feedback from test users to assess user-friendliness and design consistency.

5.1.3 Tools and Environment:

- Postman: Used to test backend API endpoints (e.g., login, credential submission, record editing).
- MongoDB Compass: Used to inspect and verify data storage and relationships in the database.

- Flutter DevTools: For debugging UI elements and state management.
- Browser DevTools (when using web output of Flutter): To monitor network requests and handle frontend errors.

5.1.4 Sample Test Cases and Results:

Test Case	Input	Expected Result	Actual Result	Status
Log in with the correct credentials	Valid email + password	Redirect to dashboard	Success	✅ Passed
Log in with incorrect credentials	Invalid password	Show error message	Error shown	✅ Passed
Submit the credential form	Valid inputs and documents	Confirmation message	Success	✅ Passed
Submit the credential form with missing fields	Empty fields	Show validation error	Error shown	✅ Passed
Admin accesses a user-only page	Admin logged in	Access denied	Access restricted	✅ Passed
Patient schedules an appointment	Available slot selected	Appointment confirmed	Success	✅ Passed
Upload an invalid file format	.exe file	File rejected	Error message	✅ Passed

Table 4. Test Cases and Results

5.1.5 Bug Fixes and Improvements

Issue	Resolution
MFA prompt not triggering after login	Fixed by verifying the token check before redirecting to the dashboard.
Admin dashboard showing all user data regardless of role	Fixed by enforcing stricter role checks on sensitive endpoints.
UI form allowed submission with empty fields	Resolved by adding client-side and server-side validation.
Alert-based errors were disruptive	Replaced with inline error messages for better UX.

Table 5. Bug Fixes and Improvements

6. Conclusions and Future Work

In conclusion, the Credential Verification System Project, developed as a mobile application, significantly enhances how healthcare worker qualifications are submitted, reviewed, and managed. By offering a user-friendly, secure, and accessible platform, the system streamlines the traditionally time-consuming credentialing process, allowing healthcare professionals and administrators to interact efficiently in real time. Features such as multi-factor authentication, structured credential submission, and mobile notifications ensure improved security, transparency, and accountability. The project reduces manual errors and paperwork, improving onboarding speed and operational efficiency across healthcare institutions. For future development, the system could be expanded to support a broader healthcare network, integrate with existing hospital and HR systems, and introduce AI-based validation tools or analytics to improve data accuracy and decision-making. Offline functionality, enhanced administrative dashboards, and compliance updates will further strengthen its value and usability in diverse healthcare environments.

6.1. Lifelong Learning

Lifelong learning plays a vital role in keeping pace with rapidly evolving technologies, especially in areas where healthcare and mobile development intersect. Developing the Credential Verification System Mobile App allowed us to gain real-world experience in building secure, scalable mobile solutions tailored to healthcare needs. Throughout the project, we strengthened our understanding of secure mobile app architecture, user authentication, data privacy best practices (such as HIPAA and GDPR), usability design for users with varying technical skills, and the importance of mobile-first development for real-time and remote access. This experience has motivated us to continue learning about new tools, frameworks, and standards in mobile health (mHealth) technology. We are committed to staying updated through courses, certifications, and project-based learning to ensure we can contribute meaningfully to the development of impactful healthcare applications. In summary, this project has reinforced the importance of continuous learning in designing mobile systems that are secure, user-centric, and adaptable to the future needs of the healthcare sector.

References:

1) Mahmoud, M. (2019, December 10). A Case Study on Hospital Management System. https://doi.org/10.13140/RG.2.2.23215.28328/1
2) Awantika. (2023, March 29). The Complete Guide to Hospital Management System. LeadSquared. https://www.leadsquared.com/industries/healthcare/hospital-management-system-hms/
3) What is Interoperability? - Interoperability in Healthcare Explained - AWS. (n.d.). Amazon Web Services, Inc. https://aws.amazon.com/what-is/interoperability/
4) Sommerville, I. (2021). Software Engineering (11th ed.). Pearson.
5) Bass, L., Clements, P., & Kazman, R. (2021). Software Architecture in Practice (4th ed.). Addison-Wesley Professional.
6) Fowler, M. (2018). Refactoring: Improving the Design of Existing Code (2nd ed.). Addison-Wesley Professional.
7) Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
8) Newman, S. (2021). Building Microservices (2nd ed.). O'Reilly Media.
9) Flutter and Mobile Development References
Napoli, M. L. (2020). Beginning Flutter: A Hands On Guide to App Development. Wiley.
10) Sharma, A. (2021). Flutter in Action. Manning Publications.
11) Biessek, A. (2019). Flutter for Beginners: An introductory guide to building cross-platform mobile applications with Flutter and Dart 2. Packt Publishing.
12) Android Developers. (2023). App Architecture Guide. Retrieved from https://developer.android.com/topic/architecture
13) Flutter Documentation. (2023). State Management. Retrieved from https://docs.flutter.dev/data-and-backend/state-mgmt/intro
14) Security and Authentication References
OWASP Foundation. (2023). OWASP Mobile Application Security Verification Standard. Retrieved from https://owasp.org/www-project-mobile-security/Seitz , J. (2022). OAuth 2.0 Simplified. Okta, Inc.
15) Ferguson, N., Schneier, B., & Kohno, T. (2010). Cryptography Engineering: Design Principles and Practical Applications. Wiley.
16) Firebase. (2023). Firebase Authentication Documentation. Retrieved from https://firebase.google.com/docs/auth
17) Database and API References
MongoDB, Inc. (2023). MongoDB Documentation. Retrieved from https://docs.mongodb.com/
18) Masse, M. (2011). REST API Design Rulebook. O'Reilly Media.
19) Lauret, A. (2019). The Design of Web APIs. Manning Publications.
20) UML and Diagram References
Fowler, M. (2003). UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd ed.). Addison-Wesley Professional.

21) Rumbaugh, J., Jacobson, I., & Booch, G. (2004). The Unified Modeling Language Reference Manual (2nd ed.). Addison-Wesley Professional.
22) Ambler, S. W. (2004). The Object Primer: Agile Model-Driven Development with UML 2.0. Cambridge University Press.