**A PROPOSED OFFERING OF A CLINIC RECORDS MANAGEMENT SYSTEM**

**FOR**

**HI-PRECISION DIAGNOSTICS – MALABON BRANCH**

A Thesis Project Presented to the

Faculty of Datamex College of Saint Adeline, Inc.

In Partial Fulfillment of the Requirements for the

Degree of Bachelor of Science in Information Technology

By:

Catubay, Mark Lawrence L.

**DESIGN DOCUMENT**

**CHAPTER I**

**INTRODUCTION**

In the 21st century, the healthcare industry is undergoing a profound digital transformation, driven by the need for greater efficiency, accuracy, and accessibility of patient information. The vast amount of sensitive data generated daily in these facilities—from patient demographics to complex laboratory results—demands a robust, secure, and systematic approach to information management. The transition from traditional paper-based record-keeping to digital systems Clinic Management Systems (CMS), is no longer a luxury but a fundamental necessity for providing high-quality patient care and ensuring operational excellence.

Hi-Precision Diagnostics is one of the leading and most trusted medical laboratories in the Philippines, renowned for its advanced technology and commitment to accurate and timely results. Its branches, including the one located in Malabon, handle a significant volume of patients daily, each requiring meticulous documentation of personal data, medical history, consultations, and prescribed treatments. The current record-keeping process, which may rely on manual ledgers, disparate spreadsheet files, or an outdated system, presents significant operational challenges. Manual systems are inherently prone to human error, such as illegible handwriting, misplaced files, and inconsistent data entry. They are also inefficient, requiring staff to spend valuable time physically searching for patient folders, which can lead to longer patient wait times and a diminished quality of service.

Furthermore, the security and integrity of paper records are a constant concern. Physical documents are vulnerable to damage from fire or water, misplacement, and unauthorized access, posing a direct risk to patient privacy and regulatory compliance with data protection laws like the Data Privacy Act of 2012 (RA 10173). The inability to easily aggregate data for analysis also means that the clinic misses opportunities to identify patient trends, manage medicine inventory effectively, and generate reports for administrative decision-making.

Recognizing these challenges, this study proposes the design, development, and implementation of a dedicated Clinic Records Management System (CRMS) for Hi-Precision Diagnostics – Malabon Branch. This system will serve as a centralized, secure, and user-friendly digital platform for managing all core aspects of the clinic's data, thereby modernizing its operations and empowering its staff to deliver superior patient care.

**Statement of the Problem**

The primary problem this project aims to address is the lack of an integrated and efficient system for managing patient and operational records at Hi-Precision Diagnostics – Malabon Branch, leading to operational bottlenecks, data security risks, and potential compromises in patient service quality.

Specifically, this study seeks to answer the following questions:

1. How can a centralized digital system improve the speed and accuracy of retrieving and updating patient records compared to the current process?
2. What features are necessary in a Clinic Records Management System to effectively manage patient information, consultation histories, and medicine inventory within a single platform?
3. How can the proposed system ensure the security, confidentiality, and integrity of sensitive patient data in compliance with privacy regulations?
4. What system architecture can be developed to provide a reliable, user-friendly, and standalone desktop application that meets the specific operational needs of the clinic staff?

**Objectives of the Study**

**General Objective:**

The main objective of this project is to design, develop, and implement a comprehensive and secure Clinic Records Management System (CRMS) for Hi-Precision Diagnostics – Malabon Branch to automate and streamline their record-keeping processes.

**Specific Objectives:**

To achieve the general objective, the proponent aims to:

1. To develop a Patient Information Module for creating, retrieving, updating, and deleting patient profiles in a centralized database.
2. To create a Consultation Management Module that allows clinic staff to record and access patient complaints, diagnoses, and treatments for each visit.
3. To design and implement a Medicine Inventory Module to track the stock levels of available medicines, including functionalities for adding new stock and updating quantities.
4. To build a secure user authentication system with password hashing to restrict access to authorized personnel only.
5. To develop a user-friendly and intuitive graphical user interface (GUI) that minimizes the learning curve for clinic staff and enhances workflow efficiency.
6. To evaluate the functionality and usability of the developed system to ensure it meets the requirements of the clinic.

**CHAPTER II**

**SYSTEM ARCHITECTURE**

This chapter describes the high-level structure of the system, including the chosen architectural pattern and the plan for deployment*.*

**Architectural Pattern**

The system follows a client-server architecture running on a single machine, packaged as a desktop application. The components are separated into three logical layers:

* **Presentation Layer (Frontend)**

A Multi-Page Application (MPA) consisting of distinct HTML, CSS, and JavaScript files for each major feature (patients.html, inventory.html, login.html). This approach provides clear separation and organization for each module. It communicates with the Logic Layer via asynchronous fetch API calls.

* **Logic Layer (Backend)**

A RESTful API built with Node.js and the Express.js framework. It handles all business logic, data validation, and request/response cycles.

* **Data Layer (Database)**

A self-contained SQLite database file (clinic.db). It is responsible for the persistence and integrity of all application data.

**Deployment Architecture**

The entire application is packaged into a single executable file using Electron.js. This framework creates a native desktop application window that loads the frontend files, while the Node.js/Express.js backend runs as a background process initiated by Electron. This allows the application to run on Windows or macOS without external dependencies or a persistent internet connection.

**CHAPTER III**

**DATABASE DESIGN**

This chapter provides a detailed blueprint of the database, including a visual diagram of table relationships and a comprehensive definition of each table's schema.

**Entity-Relationship Diagram (ERD)**

The database is comprised of four tables designed to manage users, patients, consultations, and medicines.

A screenshot of a computer screen

AI-generated content may be incorrect.

**Table Schema**

The database schema is defined as follows:

**Table: users**

|  |  |  |  |
| --- | --- | --- | --- |
| Column | Type | Constraints | Description |
| id | INTEGER | PRIMARY KEY, AUTOINCREMENT | Unique user identifier. |
| username | TEXT | UNIQUE, NOT NULL | User's login name. |
| password\_hash | TEXT | NOT NULL | Bcrypt hash of the user's password. |

**Table: patients**

|  |  |  |  |
| --- | --- | --- | --- |
| Column | Type | Constraints | Description |
| id | INTEGER | PRIMARY KEY, AUTOINCREMENT | Unique patient identifier. |
| name | TEXT | NOT NULL | Patient's full name. |
| dob | TEXT | NOT NULL | Patient's date of birth (ISO 8601 format). |
| address | TEXT |  | Patient's physical address. |

**Table: consultations**

|  |  |  |  |
| --- | --- | --- | --- |
| Column | Type | Constraints | Description |
| id | INTEGER | PRIMARY KEY, AUTOINCREMENT | Unique consultation identifier. |
| patient\_id | INTEGER | NOT NULL, FOREIGN KEY (patients.id) ON DELETE CASCADE | Links to the patient. |
| complaint | TEXT | NOT NULL | Patient's stated complaint. |
| diagnosis | TEXT |  | Doctor's diagnosis. |
| treatment | TEXT |  | Prescribed treatment or plan. |
| consultation\_date | TEXT | NOT NULL | Date of the consultation. |

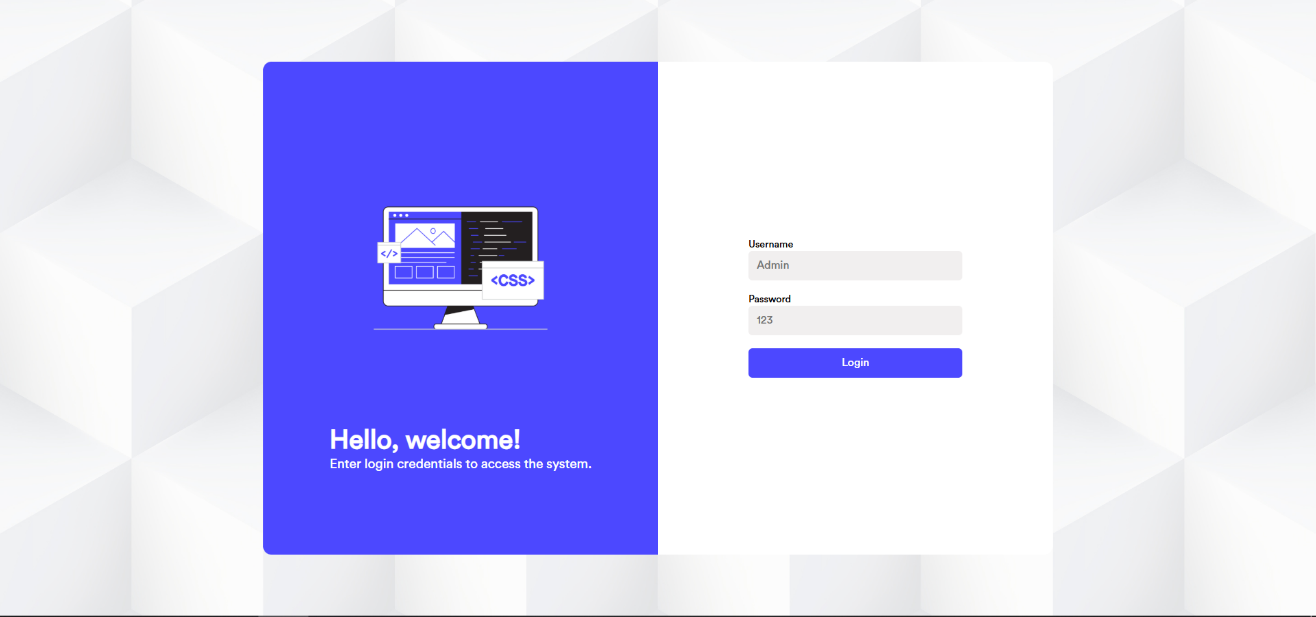
**Table: medicines**

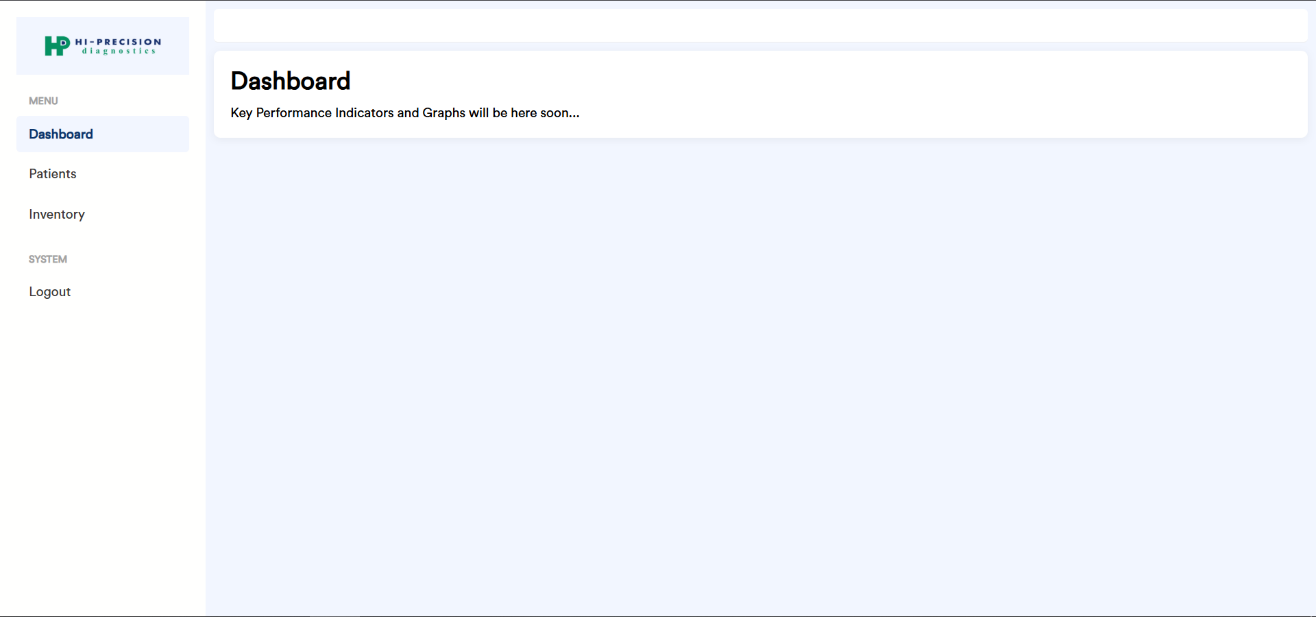
|  |  |  |  |
| --- | --- | --- | --- |
| Column | Type | Constraints | Description |
| id | INTEGER | PRIMARY KEY, AUTOINCREMENT | Unique medicine identifier. |
| name | TEXT | UNIQUE, NOT NULL | The official name of the medicine. |
| quantity | INTEGER | NOT NULL, DEFAULT 0 | Current stock level. |
| description | TEXT |  | Optional description or notes. |

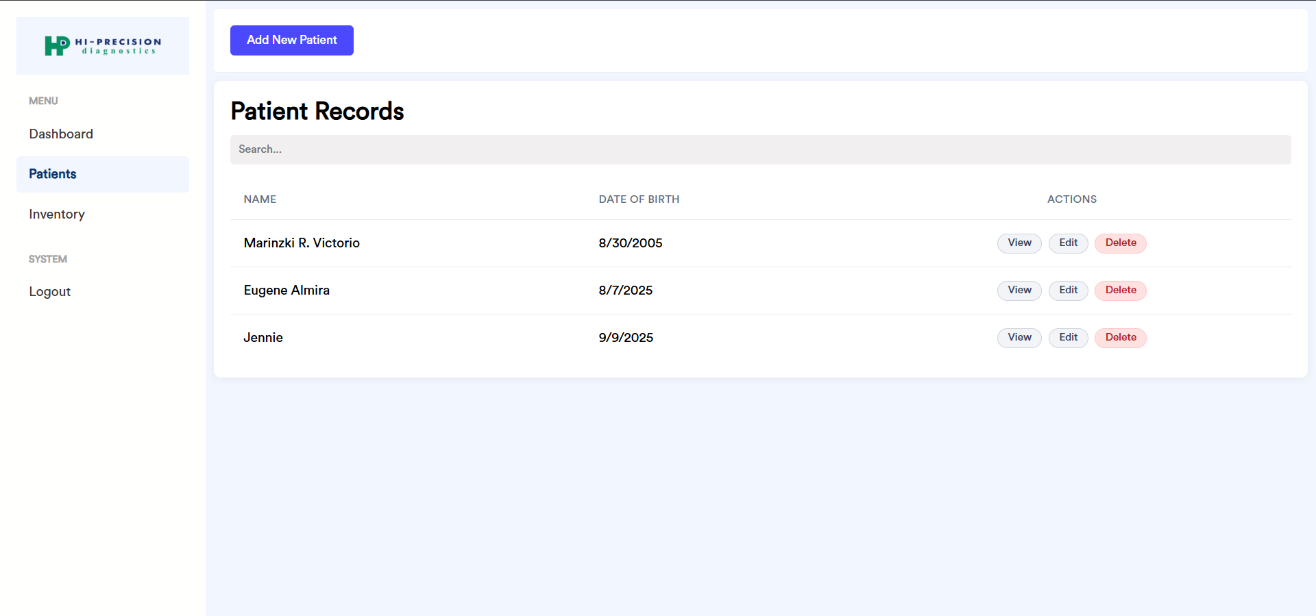
**CHAPTER IV**

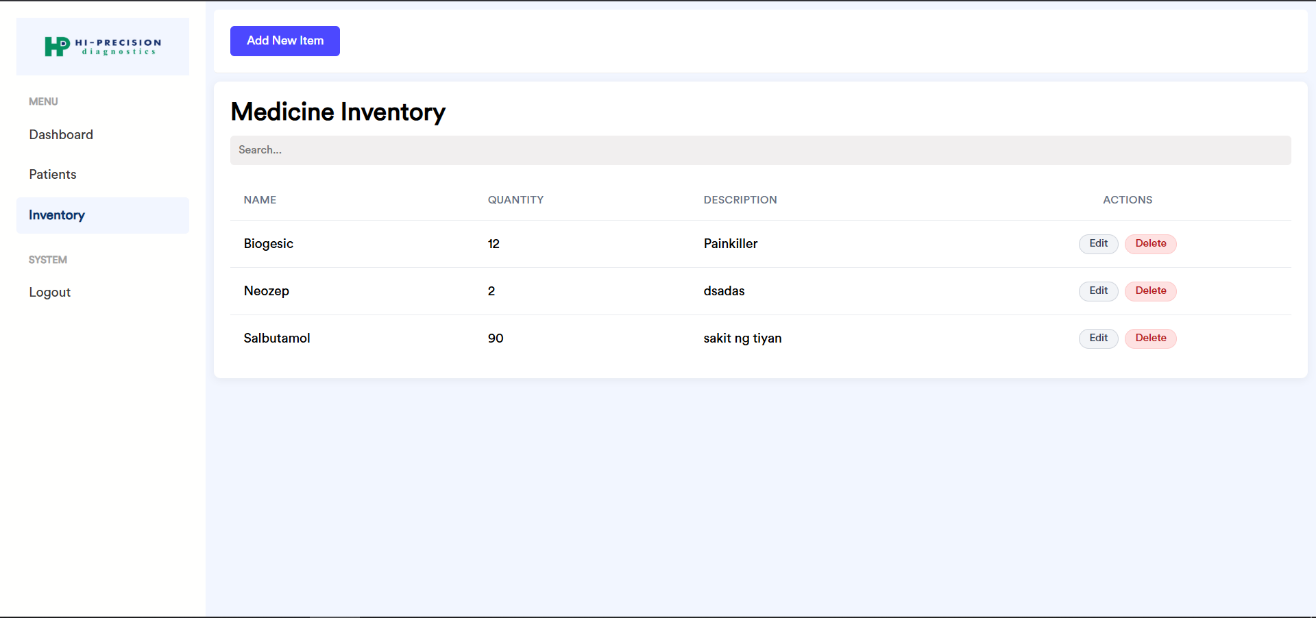
**USER INTERFACE DESIGN**

This chapter describes the visual design and user interaction model of the application, including layout, key components, and design principles.



****

****

****

**CHAPTER V**

**COMPONENT DESIGN (BACKEND)**

This chapter breaks down the backend system into its core software components and describes the responsibility of each.

**app.js**

The main server entry point. Initializes the Express application, configures global middleware (cors, express.json, express.static), and mounts the resource routers.

**database.js**

Manages the singleton connection to the clinic.db SQLite file. Exports an initDb function that creates all four application tables if they do not exist.

**routes/:**

Contains the API route definitions.

* auth.js: Defines /login and /register endpoints.
* patients.js: Defines /, /:id, and /search endpoints for patients, as well as the nested /patients/:patientId/consultations endpoints.
* medicines.js: Defines / and /:id endpoints for medicine inventory.

**controllers/:**

Contains the business logic.

* authController.js: Handles user registration and login logic, including password hashing and comparison.
* patientController.js: Manages all CRUD and search operations for patients.
* consultationController.js: Manages adding and retrieving consultations for a specific patient.
* medicineController.js: Manages all CRUD operations for the medicine inventory.

**CHAPTER VI**

**DATA FLOW DIAGRAMS (DFD)**

This chapter illustrates how data moves through the system for key processes, from user input to database storage and back to the UI.

A diagram of a medical procedure

AI-generated content may be incorrect.

**CHAPTER VII**

**SECURITY DESIGN**

This chapter details the mechanisms and strategies implemented to protect the system and its data from unauthorized access and threats.

**Authentication**

Access to the system is restricted via a mandatory login. All subsequent API requests from the frontend will be protected to ensure a valid session exists.

**Authorization**

The current design specifies a single "Clinic Staff" role with full permissions. The architecture allows for future implementation of Role-Based Access Control (RBAC) by adding a role column to the users table and implementing middleware to check permissions on API routes.

**Data Protection**

* Password Hashing: The bcryptjs library is used to perform salted, one-way hashing of all user passwords. Plain-text passwords are never stored.
* SQL Injection Prevention: All database queries are executed using parameterized statements provided by the sqlite3 Node.js library, which automatically sanitizes inputs and prevents SQL injection attacks.

**CHAPTER VIII**

**PERFORMANCE DESIGN**

This chapter addresses the strategies and considerations integrated into the Clinic Records Management System (CRMS) design to ensure it operates efficiently, responds quickly to user actions, and maintains stability under the expected operational loads of a busy diagnostic clinic. The primary goal is to deliver a smooth and responsive user experience, preventing delays that could hinder staff productivity and patient service.

**Performance Requirements and Objectives**

The performance of the CRMS is defined by its responsiveness, data processing speed, and resource consumption on a standard office computer. The system is designed for a single-user or low-concurrency environment, meaning the primary focus is on optimizing individual user interactions rather than managing simultaneous requests from numerous users. The specific, measurable performance objectives for the system are detailed below.

**Key Performance Objectives**

|  |  |  |
| --- | --- | --- |
| Metric | Target | Measurement Method |
| Application Startup Time | The application's main login window should be fully rendered and ready for user input within 5 seconds of launching the executable file. | Timed manually with a stopwatch from application launch to UI readiness. |
| User Authentication | Login and logout operations should complete in under 1 second. | Measured by logging timestamps in the backend from the moment a request is received to when the response is sent. |
| Patient Record Retrieval | Fetching and displaying the details of a single patient record should take less than 500 milliseconds. | Measured via the browser developer tools' Network tab within the Electron application. |
| Data Search Operation | Searching for a patient by name in a database containing 10,000 records should return results in under 2 seconds. | Backend timing logs for the search query execution time. |
| Data Entry & Submission | Saving a new patient record or a new consultation form should complete within 1 second. | Measured via the browser developer tools' Network tab. |
| CPU Utilization (Idle) | When the application is open but not actively being used, it should consume less than 2% of the CPU. | Monitored using the operating system's Task Manager (Windows) or Activity Monitor (macOS). |
| Memory (RAM) Footprint | The application's memory usage should not exceed 300 MB during normal, sustained operation. | Monitored using the operating system's Task Manager (Windows) or Activity Monitor (macOS). |

**Strategies for Performance Optimization**

To achieve the objectives listed above, specific design strategies have been implemented across all three tiers of the application architecture.

**Data Layer Optimization (SQLite)**

The performance of a data-driven application is heavily dependent on the efficiency of its database. The following strategies are employed:

* Database Indexing: To accelerate data retrieval operations, indexes are crucial. While primary keys are indexed by default, additional indexes will be created on columns frequently used in search queries. Specifically, an index will be placed on the name column of the patients table. Foreign keys, such as patient\_id in the consultations table, will also be indexed to speed up join-like operations and retrieval of related records.
* Efficient Query Construction: The application will avoid using SELECT \* queries, which can retrieve unnecessary data and increase processing time. Instead, queries will specify only the columns required for a particular view or operation. All database interactions will use parameterized queries, which not only prevent SQL injection but can also allow the database engine to cache and reuse query execution plans for better performance.

**Logic Layer Optimization (Node.js/Express.js)**

The Node.js backend is designed to handle business logic and data processing without becoming a bottleneck.

* Asynchronous Operations: All interactions with the database are handled using asynchronous functions (async/await). This ensures that the Node.js event loop is not blocked while waiting for a database query to complete, allowing the application to remain responsive to other potential requests.
* Payload Management: The API endpoints are designed to return only the necessary data to the frontend. For instance, a list view of patients will only return essential fields like ID and name, while a detailed view will fetch the complete record. This minimizes the size of the JSON payloads transferred between the backend and frontend.
* Data Pagination: For views that may display a large number of records, such as the main patient list or medicine inventory, the API will implement pagination. Instead of loading all records at once, the system will fetch data in manageable chunks (a "page" of 20-50 records at a time), significantly reducing initial load times and memory consumption.

**Presentation Layer Optimization (HTML/JS/Electron)**

The user interface is designed to feel fast and fluid.

* Non-Blocking API Calls: All communication with the backend API from the frontend is performed asynchronously using the fetch API. This prevents the user interface from freezing while data is being loaded. Users will see loading indicators or spinners during these brief wait times, providing visual feedback that the system is working.
* Minimal DOM Manipulation: The JavaScript code is written to perform minimal and efficient updates to the Document Object Model (DOM). When data is updated, only the relevant parts of the page are re-rendered, rather than refreshing the entire screen, which leads to a smoother user experience.

**Performance Testing Plan**

To verify that the system meets the specified performance objectives, a series of tests will be conducted before deployment.

1. Baseline Testing - The application will be tested on a machine that meets the minimum hardware requirements. All performance metrics from Table 8.1 will be measured under normal conditions with a small amount of test data.
2. Load Testing - The database will be programmatically populated with a large volume of synthetic data (such as 10,000 patient records and 50,000 consultation entries) to simulate long-term usage. The performance tests for data retrieval, search, and submission will be re-executed to ensure the system scales gracefully and continues to meet its performance targets.
3. Resource Monitoring: During all tests, the application's CPU and memory usage will be monitored continuously to identify any memory leaks or performance degradation over extended periods of use.

**Performance Test Cases**

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case ID | Description | Metric to Measure | Expected Result |
| PT-01 | Launch the application from a closed state. | Time to Login Screen | < 5 seconds |
| PT-02 | Search for a patient by name in a database with 10,000 records. | Time to display results | < 2 seconds |
| PT-03 | Open a patient record with 20 past consultations. | Time to display patient details | < 500 milliseconds |
| PT-04 | Leave the application running idle for 1 hour. | CPU and Memory Usage | CPU < 2%, Memory < 300 MB |
| PT-05 | Add 50 new medicine records sequentially. | Average time per submission | < 1 second per record |

**CHAPTER IX**

**ERROR HANDLING AND LOGGING**

This chapter describes the system's strategy for managing errors, logging important events, and communicating issues to the user. A robust error handling and logging mechanism is essential for maintaining system stability, diagnosing problems, and providing a non-disruptive user experience. The strategy is designed to be proactive in catching errors and clear in its communication.

**Error Handling Mechanisms**

The system employs a two-pronged approach to error handling, with distinct strategies for the backend API and the frontend user interface.

**Backend (Server-Side) Error Handling**

The Node.js/Express.js backend is the central point for data validation and business logic, making it the primary source of potential errors. A centralized error-handling middleware is implemented in app.js to catch all exceptions, both synchronous and asynchronous. This ensures that no error crashes the server process and that a consistent, structured response is always sent to the client.

The API uses standard HTTP status codes to communicate the nature of the error. All error responses are formatted as a JSON object with a single error key, providing a predictable structure for the frontend to parse.

**HTTP Status Codes and Error Responses**

|  |  |  |  |
| --- | --- | --- | --- |
| Status Code | Meaning | Example Scenario | Sample JSON Response |
| 400 Bad Request | The client sent invalid data, such as a missing required field or incorrect data format. | A user tries to create a new patient without providing a name. | {"error": "Patient name is required." } |
| 401 Unauthorized | The user is not authenticated and is trying to access a protected resource. | A user's session token is missing or invalid when requesting patient data. | {"error": "Authentication failed." } |
| 404 Not Found | The requested resource does not exist on the server. | The client requests details for a patient ID that is not in the database. | {"error": "Patient with ID 123 not found." } |
| 409 Conflict | The request could not be completed due to a conflict with the current state of the resource. | A user attempts to create a new user or medicine with a username/name that already exists. | {"error": "Username 'staff01' is already taken." } |
| 500 Internal Server Error | An unexpected condition was encountered on the server, such as a database connection failure or a programming bug. | The database file is corrupted, or an unhandled exception occurs in a controller. | {"error": "An unexpected server error occurred." } |

**Frontend (Client-Side) Error Handling**

The frontend JavaScript is responsible for catching responses from the backend and translating them into user-friendly feedback. Every fetch API call includes logic to handle both successful and unsuccessful responses.

* Network Errors - A try...catch block (or a.catch() method in a promise chain) is used to handle network-level failures, such as when the backend server is not running. In this case, a general message like "Cannot connect to the server. Please ensure the application is running correctly" will be displayed.
* HTTP Errors - After receiving a response, the code checks the response.ok property. If it is false (indicating a 4xx or 5xx status code), the frontend parses the JSON error message from the response body and displays it to the user. For validation errors (400), the message will be shown near the relevant form field. For all other errors, a modal alert or a toast notification will inform the user of the issue.

**Logging Requirements and Specifications**

Logging is critical for developers to debug issues and monitor the application's health. Given the standalone nature of the desktop application, all logs are directed to the standard console, which can be viewed through Electron's developer tools during development or from a terminal if the application is launched from the command line.

Two levels of logging are defined:

1. INFO - Used for tracking normal application events and milestones. This helps in understanding the application flow during runtime.
   * Examples: "Server listening on port 3000", "Database connection successful", "User 'admin' logged in successfully."
2. ERROR - Used for capturing any unexpected errors, exceptions, or failed operations that require attention. Error logs will include a stack trace when available to facilitate rapid debugging.
   * Examples: "Failed to execute query: [SQL Query] - Error: [Error Message]", "FATAL: Unhandled exception in /api/patients/:id route", "Database file clinic.db not found."

**CHAPTER X**

**THIRD-PARTY INTEGRATIONS**

This chapter lists the external services, libraries, and frameworks that the Clinic Records Management System depends on. The system does not integrate with external APIs for data exchange but relies on a carefully selected set of open-source software packages to accelerate development, ensure security, and provide core functionality. These dependencies are managed through the Node Package Manager (NPM).

**Core Platform and Runtime**

* Node.js - The primary backend runtime environment. It allows the system to run JavaScript on the server-side, providing an asynchronous, event-driven architecture that is well-suited for building fast and scalable API services.
* Electron.js - The foundational framework used to package the entire web application (HTML, CSS, JavaScript frontend, and Node.js backend) into a native, cross-platform desktop application. It provides the native window, menu, and system integrations.

**Third-Party Libraries (NPM Packages)**

The following table details the key third-party libraries used in the backend of the application, as defined in the project's package.json file. Each library plays a specific and crucial role in the system's operation.

Table 10.1: Third-Party NPM Libraries

|  |  |  |
| --- | --- | --- |
| Library Name | Version | Purpose and Role in the System |
| Express.js | ^5.1.0 | Serves as the foundational framework for building the backend RESTful API. It provides a robust set of features for routing, middleware management, and handling HTTP requests and responses, forming the backbone of the entire logic layer. |
| sqlite3 | ^5.1.7 | The Node.js driver that enables communication between the application's logic layer and the SQLite database file (clinic.db). It is used to execute all SQL queries for creating, reading, updating, and deleting data from the system's tables. |
| bcryptjs | ^3.0.2 | A critical security component used for hashing user passwords. It implements the bcrypt algorithm to create a salted, one-way hash of each user's password before it is stored in the users table, preventing plain-text storage and protecting user credentials. |
| cors | ^2.8.5 | An Express.js middleware that enables Cross-Origin Resource Sharing. While the application runs locally within Electron, this middleware is essential during development for allowing the frontend (served on one origin) to communicate with the backend API (running on another port). |

**CHAPTER XI**

**DEPLOYMENT PLAN**

This chapter outlines the process for packaging the CRMS into a standalone desktop application and deploying it onto an end-user's computer within the clinic.

**Overview of the Deployment Process**

The deployment goal is to provide the clinic with a single, easy-to-install file that sets up the entire application without requiring manual installation of dependencies like Node.js or SQLite. This is achieved using the electron-builder tool, which is configured in the project's package.json.

**Deployment Steps**

1. **Build the Application:**
   * The developer will run the build script from the project's root directory: npm run dist.
   * electron-builder will read the configuration in package.json, bundle all application source code (HTML, CSS, JS), the Node.js backend, and all node\_modules dependencies into a distributable format.
   * The output will be a Windows installer executable (e.g., CRMS-Thesis-Setup-1.0.0.exe) located in the dist directory.
2. **Distribute the Installer:**
   * The generated .exe installer file will be delivered to the client (Hi-Precision Diagnostics – Malabon Branch) via a USB flash drive or a secure digital download link.
3. **End-User Installation:**
   * The clinic staff will run the installer on the designated computer.
   * The NSIS installer wizard (as configured in package.json) will guide the user through the installation, allowing them to choose an installation directory.
   * The installer will copy all necessary application files to the chosen directory, create a desktop shortcut, and add an entry to the Start Menu.

**Hardware and Software Requirements (End-User Machine)**

* **Operating System:** Windows 10 or newer.
* **Processor:** 1.5 GHz or faster.
* **RAM:** Minimum 4 GB (8 GB recommended for smoother performance).
* **Storage:** At least 500 MB of free disk space for the application and its data.

**Configuration Management and Version Control**

* **Version Control:** All source code is managed using Git.
* **Configuration:** The application requires no manual configuration post-installation. The database file (clinic.db) is automatically created in the user's local application data folder (%APPDATA%) upon the first run, ensuring user data is stored separately from the application's installation files. This also ensures that user data is preserved even if the application is uninstalled and reinstalled.

**CHAPTER XII**

**MAINTENANCE AND SUPPORT**

This chapter provides the framework for the ongoing maintenance of the Clinic Records Management System (CRMS) after its initial deployment. The primary goal of this plan is to ensure the system remains stable, secure, and functional while minimizing any disruption to the daily operations of the Hi-Precision Diagnostics – Malabon Branch.

**Guidelines for System Maintenance and Support**

The maintenance philosophy for the CRMS is proactive and scheduled. All updates and support activities are designed to be performed with clear communication and planning to prevent interference with patient care.

* **Software Updates -** These include patches for bug fixes, security updates for third-party libraries (like Express.js or Electron), or the introduction of new, agreed-upon features. Updates will be delivered as new versions of the installer executable.
* **Data Backup -** Because the system uses a local SQLite database file (clinic.db), the clinic is responsible for its own data integrity. It is strongly recommended that the clinic staff perform regular backups by making a copy of the clinic.db file and storing it on a separate, secure device (like an external hard drive or a secure network share) at the end of each business day.
* **Support Availability -** The Project Proponent will serve as the primary point of contact for all technical support issues, including bug reports, questions about functionality, and coordinating updates.

**Procedures for Handling Software Updates, Patches, and Bug Fixes**

The update process is designed to be straightforward and minimally invasive. It will not require the clinic to halt operations for an entire day. The downtime is limited to the few minutes required to install the new version on the computer.

**Update Workflow**

1. **Issue Identification & Development -** When a bug is reported by the clinic or a new feature is required, the Project Proponent will perform all development and testing on a separate development machine. This has **zero impact** on the live system at the clinic.
2. **Packaging -** Once the update is complete and tested, the Proponent will package the new version of the application into a new installer file. This new file will be clearly versioned
3. **Communication & Scheduling -** The Proponent will contact the clinic to:
   * Inform them that an update is ready.
   * Provide a "changelog" detailing what has been fixed or added.
   * Schedule a time for the update.
4. **Scheduling the Update (Addressing Downtime)**
   * **Recommendation -** All updates must be scheduled during non-operational hours to ensure no active data entry is interrupted.
   * **Ideal Times:**
     + At the end of the business day after the last patient has been processed.
     + During a weekend when the clinic is closed.
     + On a public holiday.
   * **Impact:** This scheduling strategy ensures that clinic operations are not halted. Staff will leave work with the old version and return to the new, updated version without any interruption to their workflow.
5. **On-Site Deployment Process**
   * **Step 1 - CRITICAL - Backup the Database:** Before any changes are made, the clinic staff or proponent must create a backup copy of the clinic.db file.
   * **Step 2 - Close the Application:** The CRMS application must be fully closed on the computer before running the update.
   * **Step 3 - Run the New Installer:** The new .exe installer is executed. It will overwrite the existing application files with the updated versions. **This process does not affect the clinic.db database file**, as it is stored separately in the user's local data folder (%APPDATA%), not in the application's installation directory.
   * **Step 4 - Verification:** After the installation is complete, the application is launched to verify that it opens correctly, the version number is updated (if applicable), and all existing patient data is present and intact.

**Escalation Process for Resolving Issues**

To ensure timely resolution of any problems, the following support structure is established:

* **Level 1 Support (Primary Contact) -** For any bugs, errors, or operational questions, the clinic staff should first contact the Project Proponent.
  + **Contact Person -** Catubay, Mark Lawrence L.
  + **Contact Method -** Email (marklawrencecatubay@gmail.com) or Phone (09058591299).
  + **Required Information -** When reporting an issue, staff should provide a clear description of the problem, any error messages displayed, and the steps taken that led to the issue. Screenshots are highly encouraged.
  + **Response Time -** The Proponent will acknowledge receipt of the issue within 24 business hours.
* **Level 2 Support (Academic Oversight) -** In the event the Project Proponent is unreachable for an extended period, the Project Advisor can be contacted for assistance.

**CHAPTER XIII**

**REVISION HISTORY**

This chapter tracks the changes and updates made to this design document over time.

* Log of changes made to the document, including dates and descriptions of changes.

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Changes |
| 1.0 | August 27, 2025 | Catubay, Mark Lawrence L. | Initial draft of the Design Document. |
| 1.1 | September 3, 2025 |  | Deployment Plan, Maintenance and Support, Appendix left |
| 1.2 | September 8, 2025 |  | Refined DFD, Deployment Plan, Maintenance and Support |

**CHAPTER XIV**

**APPENDIX**

This chapter contains supplementary materials, diagrams, and other information that supports the main body of the document.

* Any additional supporting documentation (e.g., diagrams, reference materials).

Supporting documentation includes the Software Development Plan (SDP) and the Requirement Specification Document (RSD).

(TO BE DONE)