CardioLink

DEVELOPER MANUAL

Amalia Rial Plaza
Rodrigo Fernández Sánchez
Lorena Cano Díaz-Maroto
Carmen Caballero Herreros
Anastasia Ricchiuti

1-12-2025

Index

1. Introduction

1.1 Project overview

This project is aimed at the development of an application for the telemonitoring of patients with chronic conditions, enabling continuous supervision from their home environment through the use of networking algorithms and physiological sensing technology. By employing BITalino devices, which can acquire ECG (electrocardiogram) and EDA (electrodermal activity) data, the system enables early detection of physiological alterations that may indicate clinical deterioration, allowing faster response and intervention.

When the patients initialize the application, they will be asked to introduce their personal information to create an account or log into an existing one. The patient then will have the option to start recording or to see their previous recordings already checked and filled with their diagnosis by their doctor. If they choose to start a new recording, once they finish, the patient will be required to fill their symptoms from a pre-selected list. After completing all required fields and the monitoring session, the collected data is automatically transmitted to the Server, where it is processed and stored in the system's database together with the date and time of acquisition.

For doctors, after they have launched their application, they gain access to an interactive menu that offers two main functionalities. The first allows them to search for a patient in a list of all patients registered in the system to visualize the personal information of the patient and inspect their medical data in detail, including the full ECG and EDA recordings presented graphically, alongside all symptom information previously reported by the patient. The second allow them to see a list of the recently finished recordings of the patients where he needs to add a diagnosis.

The structure that allows the system to operate efficiently is composed of four independent applications, Patient, Doctor, Administrator and Server, each fulfilling a specific role within the telemedicine workflow. The Server functions as the central component, managing connections from multiple clients simultaneously, storing and retrieving data, and ensuring stable communication. An administrator interface allows authorized users to shut down the Server securely, ensuring controlled operation of the entire system.

1.2 Architecture

This Java application consists of a client component, used by patients, doctors, or administrators, which connects to a central server and a database responsible for storing all data.

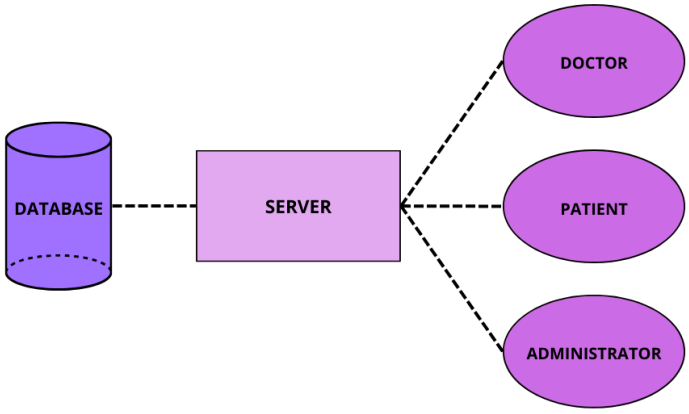


Figure 1. *Scheme of the architecture of the application*

1.3 Application Requirements

For the application to function properly, the firewall on the machine running it should be disabled, and Bluetooth must be enabled to allow communication with the BITalino device.

The libraries used in this project are listed in the tables below. For the patient application, Table [1] includes the libraries required to establish communication with both the Server and the BITalino device, as well as to acquire and transmit data. For the doctor application, Table [2] presents the libraries responsible for connecting to the Server, retrieving information, and generating the visual representation of the ECG and EDA signals. In the administrator application, Table [3] outlines the libraries used to remotely shut down the server. Finally, Table [4] contains the libraries used by the server, which are primarily dedicated to managing the database connection.

Libraries/ Dependencies	**Purpose**
java.io	Control of file handling and data exchange between server and clients.
javax.swing	To build all application windows and interface components.

java.awt	Support the basic GUI elements and layout handling used across all application interfaces.
java.time	Handle dates and times.
java.util	Read and handle user generated inputs.
javax.bluetooth	Establish the Bluetooth link between the BITalino device and the computer.
javax.microedition	Set up the Bluetooth connection and manage the data exchanged over it.
org.jfree	Generate a visual representation of the selected parameters.
java.net	Establish the client-server connection using TCP sockets.
java.text	Handle date formatting and parsing within the application.
Bluecove-2.1.2.jar	To establish and manage Bluetooth communication with the BITalino device.
Commons-logging-1.2.jar	Provide a lightweight logging framework used internally by dependent libraries.
Jfreechart-1.5.4.jar	Generate charts for visualizing ECG and EDA signals within the application.

Sqlite-jdbc-3.8.7.jar	Enable the application to connect to and interact with the SQLite database.
zip.GZIPOutputStream	Comprime datos en formato GZIP mientras se escriben en un flujo de salida.

Table 1. *Libraries and Dependencies of Patient Client.*

Libraries/ Dependencies	**Purpose**
java.io	Control of file handling and data exchange between server and clients.
javax.swing	To build all application windows and interface components.
java.awt	Support the basic GUI elements and layout handling used across all application interfaces.
java.lang	Provide core language features essential for application logic and runtime operations.
java.net	Establish the client-server connection using TCP sockets
java.time	Handle dates and times.
java.util	Read and handle user generated inputs.
javax.bluetooth	Establish the Bluetooth link between the BITalino device and the computer.

javax.microedition	Set up the Bluetooth connection and manage the data exchanged over it.
java.text	Handle date formatting and parsing within the application.
zip.GZIPInputStream	Descomprime datos en formato GZIP mientras se leen desde un flujo de entrada.

Table 2. *Libraries and Dependencies of Doctor Client.*

Libraries/ Dependencies	**Purpose**
java.io	Control of file handling and data exchange between server and clients.
javax.swing	To build all application windows and interface components.
java.awt	Support the basic GUI elements and layout handling used across all application interfaces.
java.net	Establish the client-server connection using TCP sockets.

Table 3. *Libraries and Dependencies of Admin Client.*

Libraries/ Dependencies	**Purpose**
javax.crypto	Provides the framework for cryptographic operations, enabling

	data encryption, decryption, and key generation.
java.security	Provides the framework for access control, digital signatures, and managing keys and certificates.
java.sql	Manage the database.
org.springframeworksecurity. crypto.bcrypt.BCrypt	Provides secure password hashing and password checking.
java.net	Establish the client-server connection using TCP sockets.
java.time	Handle dates and times.
java.util	Read and handle user generated inputs.
java.text	Handle date formatting and parsing within the application.
Bytes-1.5.0.jar	Manage low-level byte operations required for communication with external devices.
zip.GZIPOutputStream	Comprime datos en formato GZIP mientras se escriben en un flujo de salida.
zip.GZIPInputStream	Descomprime datos en formato GZIP mientras se leen desde un flujo de entrada.

Table 4. *Libraries and Dependencies of Server.*

1.4 Parameters and Processing

To exchange information between the Server and the Client, the system prioritizes flexibility and ease of data manipulation. Consequently, the majority of the parameters transmitted across the network are serialized as String objects. Although this approach offers less strict encapsulation compared to custom objects, the String type was selected because it allows for seamless interpretation and conversion between the various formats required by the application logic.

However, specific exceptions are made for unique identifiers, such as the IDs for diagnosisFiles. These are transmitted directly as primitive integers (int). This design choice prevents the need for unnecessary type casting and eliminates the risk of format rejection (such as NumberFormatException) during database operations.

All messages related to recording data are transmitted in compressed form: this includes the transmission of recording files, the storage of recorded signal fragments in the database (since ECG and EDA segments with real-valued samples occupy substantially more space than what can be streamed directly), and also the moments when the Doctor accesses and views an ongoing or past recording. Because the volume of information involved in these operations exceeds what a standard OutputStream can support, the Server compresses all outbound data using a GZIPOutputStream wrapped around a ByteArrayOutputStream, ensuring that all content is encoded in UTF-8 before transmission. The Doctor Client then receives and decompresses the GZIP payload, reconstructing the full dataset for local processing. This method ensures efficient and reliable large-scale data transfer within the constraints of the communication channel.

Regarding process synchronization, the system relies on a handshake protocol:
State Signaling: Both the Server and the Patient Client exchange specific String messages to indicate that a process has either successfully initiated or finalized.
Validation: The application utilizes the String.equals() method to verify these control messages. If the received string matches the expected protocol command, the program flow continues to the next stage.

Error Handling: If the control message is not received correctly (indicating a failure in the start or end of a process), the system triggers an internal exception. This error is immediately caught and presented to the user through a GUI Error Window, ensuring the user is informed of the transmission failure.

2. Installation

To install the applications, you need to have Java 25 installed on your computer (you can download it from here: https://www.oracle.com/es/java/technologies/downloads/). Within the project folder, navigate to the *out/artifacts* directory and locate the folder containing the JAR file. You will find two files: the .jar file and the *run.bat* file. If double-clicking the .jar file does not work, we have created an executable *run.bat* file that allows you to run the program without having to access the command line. Note that the *run.bat* file will not work unless the .jar file is also downloaded.

On the server, the database is automatically created. However, if you already have the database set up in the *out/artifacts* directory, the application will connect directly to it. (We highly recommend keeping our database since we have kept recordings the doctor might want to see using our View Recording feature).

3. System Features

3.1 List of Actions by Application

The system provides each type of user with a distinct set of actions. In the case of the patient [5], most actions are oriented toward generating or submitting new data to the application, such as creating recordings and adding symptoms. In contrast, the doctor's actions [6] are mainly focused on accessing and reviewing the information stored in the server's database, including patient profiles, diagnosis files, and physiological recordings.

Action	Description
Establish connection	Executed when the user inputs both the IP and the port
Register	User press the "register" button and introduces his username, name, surname, password, DNI, birthday, email, sex, phone number, health Insurance number and emergency contact.

Login	User press the “login” button and introduces his username and password.
Make a new recording	User press “record BITalino signal” and the BITalino starts recording. When they stop the recording, they have to report the symptoms.
Report symptoms	After the recording is done, the patient is asked to select the symptoms suffered throughout the recording
Log out	User press “Log out” button to exit the application.

Table 5. *Actions of Patient*

Action	**Description**
Establish a connection	Executed when the user inputs both the IP and the port
Register	User press the “ Doctor sign up” button and introduces his username, name, surname, password, DNI, birthday, email, sex, specialty and license number.
Login	User press the “login” button and introduces his username and password.
Search patient	User press “Search Patient” button and can visualize a list of all the patients.

Visualize information of a patient	User select a patient of the list to see all his information (name, date of Birth, Insurance number and sex) and his diagnosis History.
View diagnosis Files	User selects a diagnosis from the diagnosis History list to see all the information about it.
Download diagnosis File	User press "Download File" button and the diagnosis File is download in his PC.
View recording	User press "View Recording" button and visualize the graph of the recording at the first 10 seconds
Visualize next 10 seconds of the recording	User press the fragment he wants to see in the table of fragments.
Download recording	User press "Download recording" and the data of the ECG and EDA are download in his PC as a .csv
Modify recently finished recording	User press "Recently Finished" button and select the record he wants to modify (introduce a diagnosis and medication).
Log out	User press "Log out" button to exit the application

Table 6. *Actions of Doctor*

Action	**Description**
Enter IP address	User introduces the IP address of the server computer and check if it is a valid IP address.

Enter the port number	User introduces the port number of the server computer and check if it is a valid port (between 1024 and 65535 ports).
Close the server	Server is closed, so the application stops working.
Disconnect from the server	User press “Disconnect” button to exit the application

Table 7. *Actions of Admin*

3.2 Communication Protocol

We chose to use TCP (Transmission Control Protocol) because reliability is essential in healthcare applications. If any packet is lost, TCP detects it and retransmits the data, ensuring that no information is missing. It also preserves the original order of transmission, which is crucial for time-series signals such as ECG and EDA, so that sequences are received correctly.

While TCP is slower than UDP, the latter provides no guarantee of delivery, retransmission, or ordering, making it unsuitable for medical telemetry where missing or corrupted values could lead to serious consequences. Since our application does not require real-time live viewing of the monitoring sessions, there is no practical advantage to using UDP, and TCP remains the safer and more robust choice.

The communication protocol is a custom, text-based protocol over TCP sockets, with one central server (ServerThread) and three clients: PatientSwing, DoctorApplicationGUI, and AdminSwing. Each client identifies themselves with a first message, and then talks to a dedicated server-side handler thread (ServerPatientThread, ServerDoctorThread, ServerAdminThread) using commands like "LOGIN", "START", "SHUT DOWN", etc.

3.2.1 Global architecture and handshake

The server opens a ServerSocket on a chosen port.

For each incoming Socket:

1. It creates DataInputStream / DataOutputStream.

2. Reads the first UTF string: the client type.
    - "PATIENT" → create ServerPatientThread
    - "DOCTOR" → create ServerDoctorThread
    - "ADMIN" → create ServerAdminThread

Starts the corresponding Runnable in its own thread, so all clients are handled concurrently.

From that moment on, everything is a request–response style exchange: client sends a command (UTF string) + parameters; server processes it in the appropriate state and answers with one or more decorators (UTF, int, boolean, etc.).

3.2.2. Patient Protocol

Patients can authenticate, start/stop physiological recordings, submit symptoms and view their diagnosis files.

In the server side the ServerPatientThread class implements an state machine in which each incoming command is routed depending on *state*.

```
enum State {
    AUTH,                 // SIGNUP / LOGIN / QUIT
    MAIN_MENU,            // START-STOP-SYMPTOMS / LOG_OUT / QUIT

}
```

AUTH

After identification:

- Patients may sign up, log in, or quit.
- Server replies with success/failure messages and transitions to the correct state.

RECORDING SESSION

A recording consists of ECG/EDA fragments sent repeatedly by the patient.

1. Patient sends “START” → server creates a new diagnosis file and responds ready.
2. Patient streams data fragments every 10 seconds; server stores each fragment and confirms reception.
3. When patient sends “STOP”, the server expects the final fragment, confirms, and finalizes the session.
4. Server requests symptoms; patient sends selected symptoms; server stores them.

DIAGRAM

```
sequenceDiagram
    PatientSwing as P
    Server as S

    P->>S: Connect + ”Patient”
    S->>P: AUTH options

    P->>S: LOGIN / SIGNUP
    S->>P: Result (OK/ERROR)
    S->>P: MAIN_MENU

    P->>S: START
    S->>P: Ready to record

    loop Fragments
        P->>S: Fragment
        S->>P: Confirm
    end

    P->>S: STOP
    S->>P: Confirm stop

    S->>P: Request symptoms
    P->>S: Symptom list
    S->>P: Symptoms saved
```

3.2.3. Doctor Protocol

Doctors authenticate, search assigned patients, view patient information, inspect ECG/EDA recordings fill out diagnosis files.

In the server side the ServerDoctorThread class implements an state machine in which each incoming command is routed depending on *state*.

```
private enum State {
    AUTH,                 // SIGN_UP / LOG_IN / QUIT
    DOCTOR_MENU,          // main menu after login
    SEARCH_PATIENT,       // doctor searches/chooses a patient
    VIEW_PATIENT,         // info of one patient
    VIEW_DIAGNOSISFILE,   // list/detail of diagnosis files
    VIEW_RECORDING,       // recording and fragments
    RECENTLY_FINISH,      // list of recently finished diagnosis
    COMPLETE_DIAGNOSISFILE // finishing a diagnosis file
}
```

Doctor can move up/down these levels using standard commands (search, view, back, etc.).

AUTH

Doctor logs in or signs up; server validates credentials and transitions to DOCTOR_MENU.

SEARCHING PATIENTS

Doctor requests the list of patients.

Server responds with the patients insurance number.

Doctor selects one → server sends patient details + diagnosis history.

VIEW DIAGNOSIS FILE

Doctor can:

- Open a diagnosis file
- Review symptoms
- Inspect textual information

COMPLETE A DIAGNOSIS FILE FROM RECENTLY FINISHED RECORDING

Doctor can:

- Select a recently finished recording
- View the patient information and symptoms reported
- View the recording
- Fill out diagnosis and medication

RECORDING INSPECTION

Doctor may:

- Request specific fragments (e.g., previous/next)
- View the 10 second fragments of that selected timeframe.

DIAGRAM

```
sequenceDiagram
    DoctorApplicationGUI as D
    Server as S

    D->>S: Connect + "Doctor"
    S->>D: AUTH options

    D->>S: LOGIN / SIGNUP
    S->>D: Result
    S->>D: DOCTOR_MENU

    D->>S: SEARCH_PATIENT
    S->>D: Patient list

    D->>S: Select patient
    S->>D: Patient info + diagnosis list

    D->>S: Open diagnosis file
    S->>D: Diagnosis details
```

```
    D->>S: List Diagnosis files to complete
    S->>D: List of Diagnosis files uncompleted

    D->>S: Completed Diagnosis file
    S->>D: Saved

    D->>S: View recording
    S->>D: Recording fragment

    D->>S: Download recording
    S->>D: Whole recording
```

3.2.4. Admin Protocol

Admin connects exclusively to control server.

The administrator is the only one that can shut down server (which requires the correct password).

They can also disconnect from the application without affecting the server.

FLOW

1. Admin connects and identifies as "ADMIN".
2. Admin chooses one of two actions:
    a. “SHUT DOWN” + password
        i. If password is correct, server broadcasts closing message and shuts down.
    b. “EXIT”
        i. Only the admin client disconnects.

DIAGRAM

```
sequenceDiagram
    AdminSwing as A
    Server as S

    A->>S: Connect + ”ADMIN”
    S->>A: Ready

    A->>S: ”Shut down” + password alt Password correct
    S->>A: Server closing message
    S->>S: Shutdown procedureselse Wrong password
    S->>A: Incorrect password end

    A->>S: ”exit”
    S->>A: Disconnect admin client
```

4. Database Management

The server maintains all persistent data in a local SQLite database file named CardioLink.db. The schema is organized around five main tables: users, patients, doctors, diagnosisFiles and recordings (linked to a specific diagnosis file).

The users table is linked one-to-one with patients and doctors via the userId field.

Each patient can have multiple diagnosis files (patientId in diagnosisFiles), and each diagnosis file has a recording session attached to it.

Cascade deletion is enabled from patients to diagnosisFiles and from diagnosisFiles to recordings, so removing a patient automatically removes all associated clinical data.

Tables [8], [9],[10], [11] and [12] reflect the attributes of the main tables of the database.

Attribute	**Type**	**Description**
idUser	INTEGER	Primary key, auto-increment
username	TEXT	Unique login name
password	TEXT	Hashed password
role	TEXT	User role (PATIENT or DOCTOR)

Table 8. *Attributes of "users" table.*

Attribute	**Type**	**Description**
idPatient	INTEGER	Primary key, auto-increment
userId	INTEGER	FK to user (idUser)
namePatient	TEXT	Patient name

surnamePatient	TEXT	Patient surname
dniPatient	TEXT	Unique identifier (DNI)
dobPatient	DATE	Date of birth
emailPatient	TEXT	Email address (unique)
sexPatient	TEXT	Sex
phoneNumberPatient	INTEGER	Phone number
healthInsuranceNumberPatient	INTEGER	Health Insurance number
emergencyContactPatient	INTEGER	Emergency contact phone

Table 9. *Attributes of "patients" table.*

Attribute	**Type**	**Description**
idDoctor	INTEGER	Primary key, auto-increment
userId	INTEGER	FK to user (idUser)
nameDoctor	TEXT	Doctor name
surnameDoctor	TEXT	Doctor surname
dniDoctor	TEXT	Unique identifier (DNI)
dobDoctor	DATE	Date of birth
emailDoctor	TEXT	Email address

sexDoctor	TEXT	Sex
specialty	TEXT	Medical specialty
licenseNumber	TEXT	Professional license number

Table 10. *Attributes of "doctors" table.*

Attribute	**Type**	**Description**
idDiagnosisFiles	INTEGER	Primary key, auto-increment
symptoms	TEXT	Free-text description of symptoms
diagnosis	TEXT	Diagnosis provided by the doctor
medication	TEXT	Prescribed treatment
date	DATE	Date of the diagnosis file (NOT NULL)
patientId	INTEGER	FK to patients(idPatient) (ON DELETE CASCADE)
status	BOOLEAN	Status flag (finished/not finished diagnosis)

Table 11. *Attributes of "diagnosisFiles" table.*

Attribute	Type	Description
id_recording	INTEGER	Primary key, auto-increment
diagnosisFileId	INTEGER	FK to diagnosisFiles(idDiagnosisFile) (ON DELETE CASCADE)
data	TEXT	Encoded ECG/EDA recording data
sequence	INTEGER	Sequential index of the recording within the diagnosis file
anomaly	BOOLEAN	Indicates if an anomaly was detected in this recording

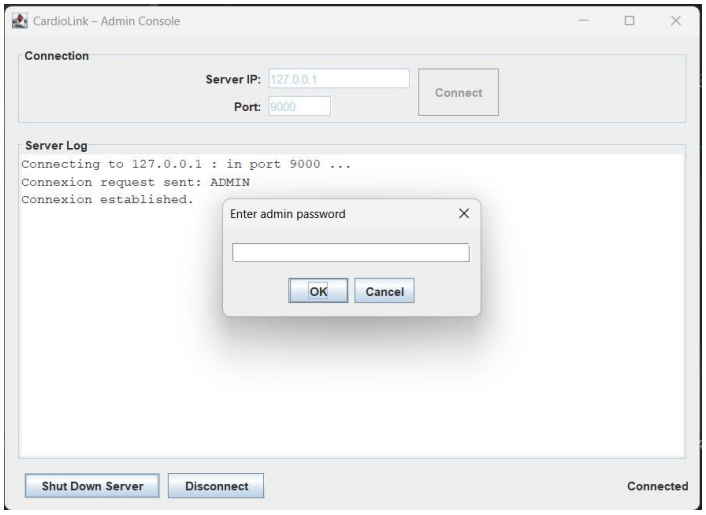
Table 12. *Attributes of "recordings" table.*

5. Server Application Manual

The server facilitates application functionality by acting as an intermediary between the clients and the database. It initializes by binding a socket to port 9000 to listen for incoming connections; if the port is unavailable, an exception is thrown. To handle concurrency, the server spawns a dedicated thread tailored to the specific client role (patient, doctor, or administrator) for each request. This multi-threaded process repeats continuously until the administrator explicitly terminates the program.

5.1 Server Shutdown

To shut down the server, the administrator needs to press the “Shut Down Server” button, then a window will appear asking for his password. To correctly shut down server, administrators have to introduce the password and press “ok” button.

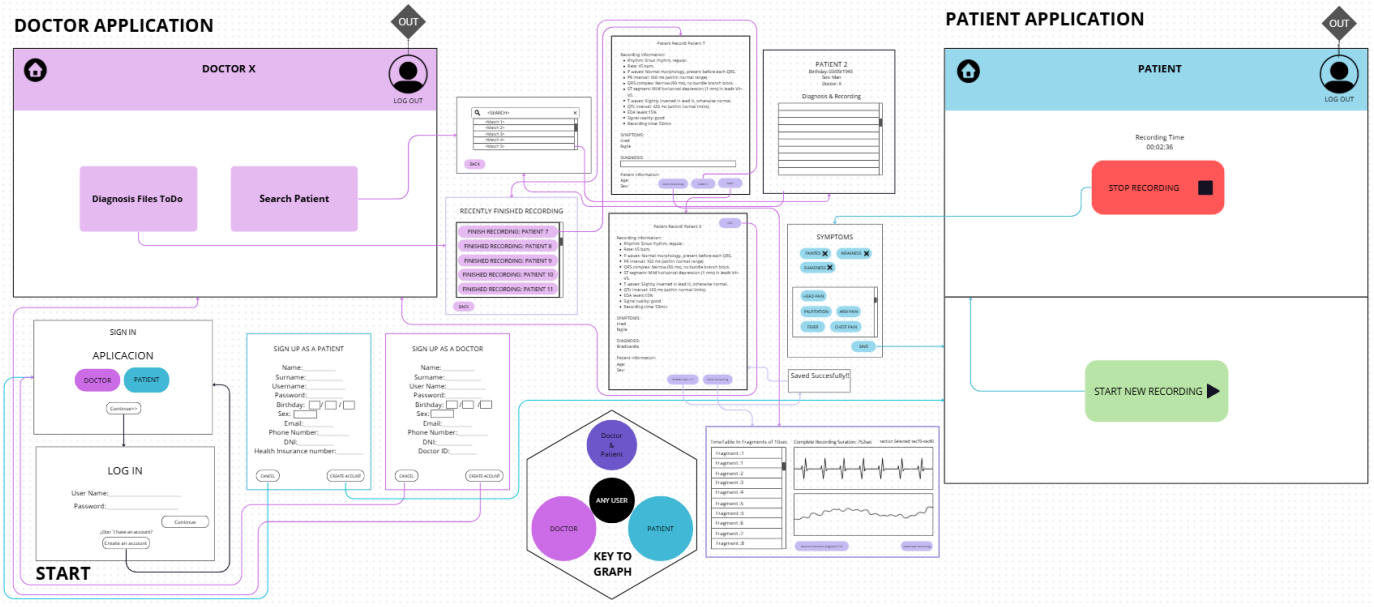


6. System Design Documentation

6.1 Mock-up

The mock-ups included in this section illustrate the visual layout and interaction flow of the graphical user interfaces for the doctor, patient, and administrator applications. They serve as design references for how users navigate through the system.

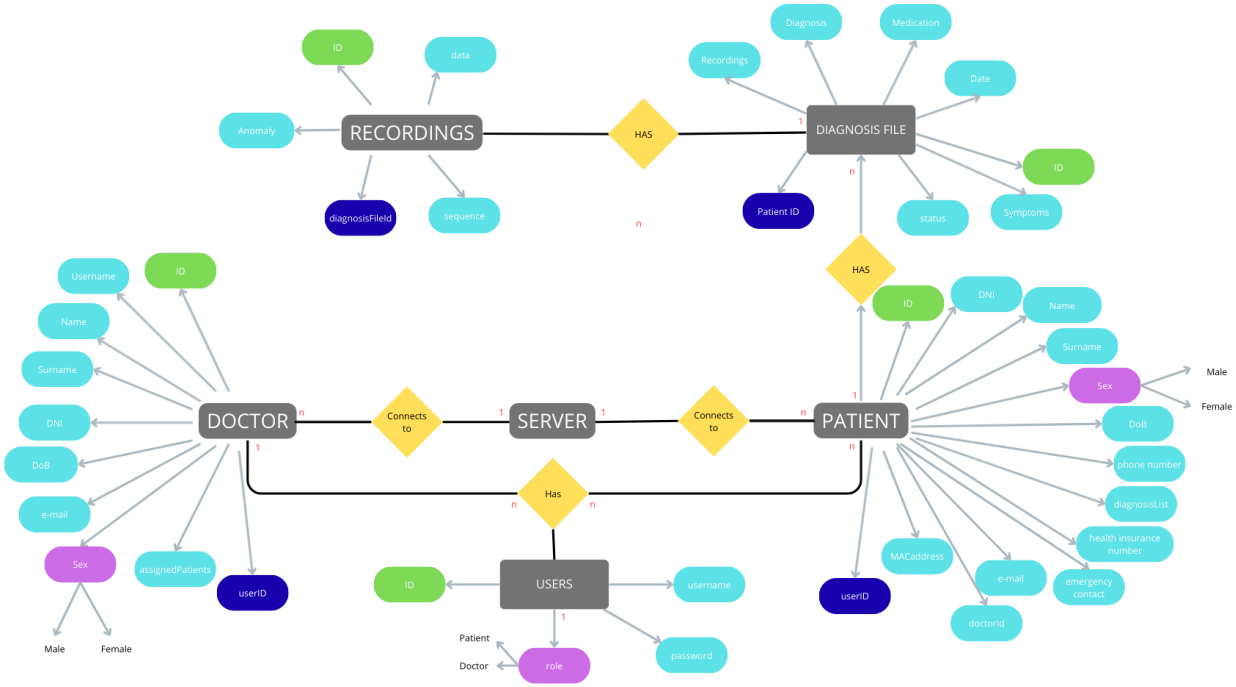
Link:
https://www.canva.com/design/DAGzV7QTLmU/Xyp3yZUhwWAir_X6Lg1ONw/edit



6.2 E-R Diagram

The entity–relationship (E-R) diagram presented in this section describes the logical data model that supports the entire system. It defines the core entities managed by the application and the relationships between them, providing developers with a clear understanding of how information is structured, stored, and linked within the database. This diagram serves as a reference for database implementation, maintenance, and future extensions, ensuring that all components of the system interact consistently with the underlying data architecture.

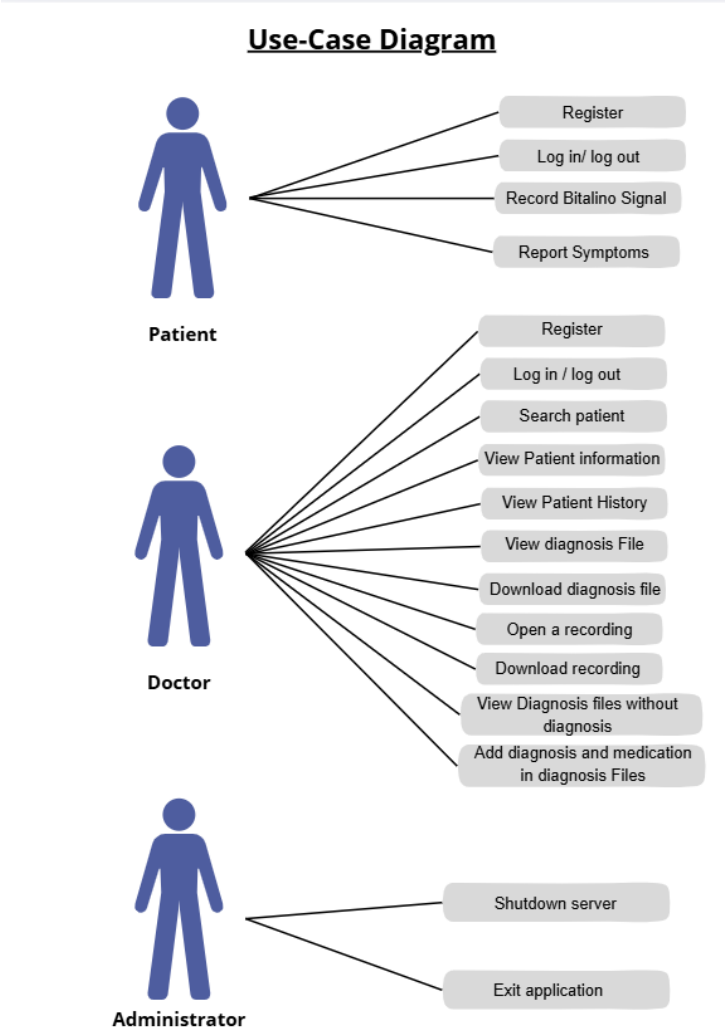
Link:
https://www.canva.com/design/DAG0Gew1ZUQ/TUyzMUTyNS16JkMKhRCznw/edit?utm_content=DAG0Gew1ZUQ&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton



6.3 Use-Case Diagram

The use-case diagram included in this section outlines the functional behavior of the system from the perspective of its external actors. It identifies the actions available to patients, doctors, and the server administrator, and illustrates how each actor interacts with the application's core features. This diagram provides developers with a high-level view of the system's expected functionality, serving as a foundation for understanding requirements, designing components, and validating the overall behavior of the platform.

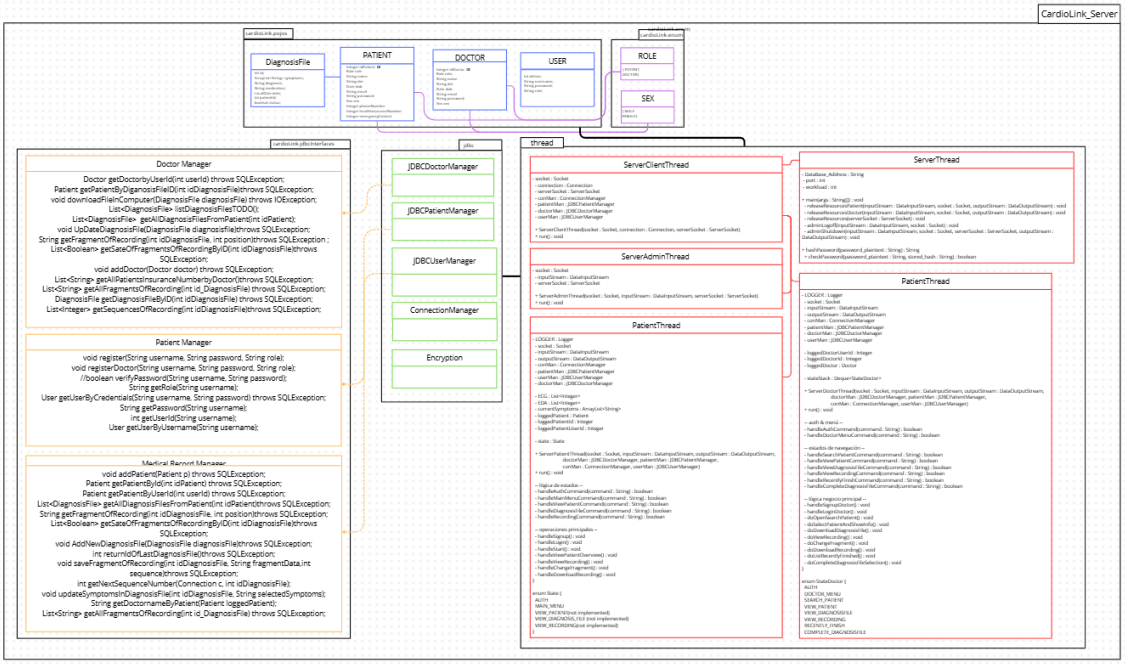
Link:
https://www.canva.com/design/DAG0GNKAkS4/yZOVSDYJHJGYDRyespPQNw/edit?utm_content=DAG0GNKAkS4&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

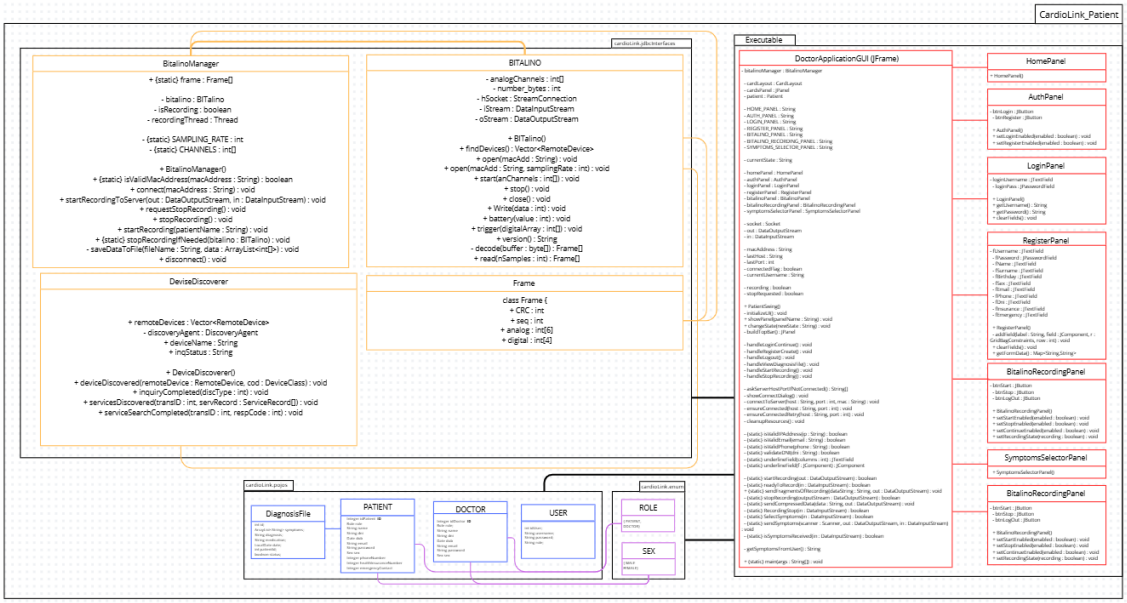


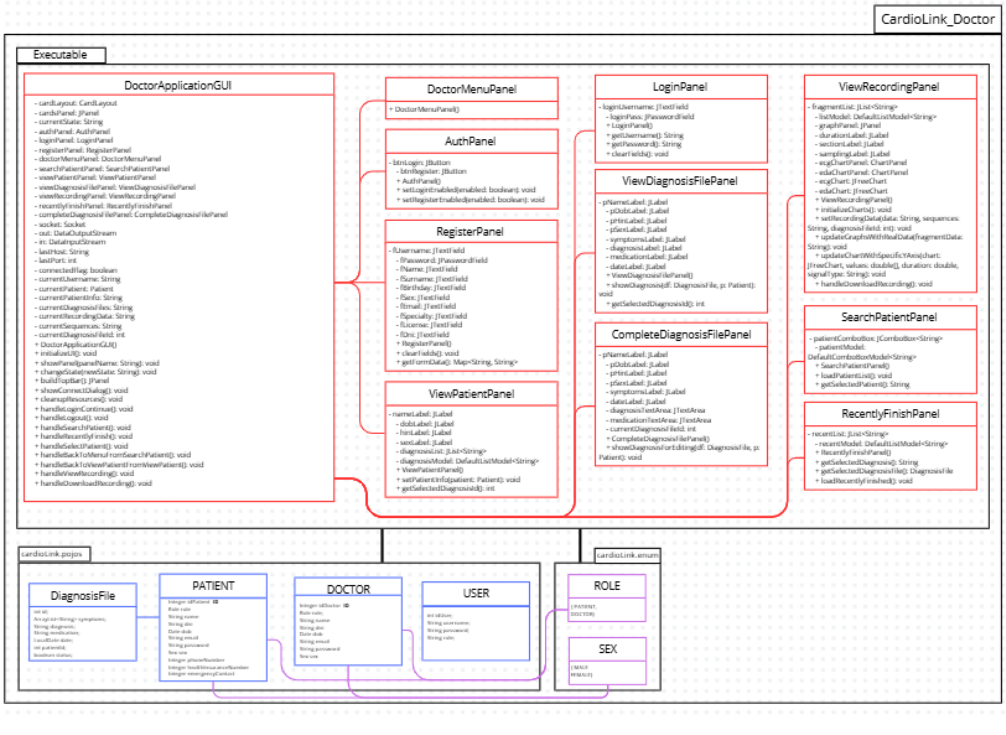
6.4 UML

The UML (Unified Modeling Language) diagrams in this section provide a standardized visual representation of the system's structure, components, and interactions. These diagrams are essential for understanding the system's design, helping developers visualize how different parts of the application collaborate and interact. By using UML, the design is clearly communicated, ensuring consistency across the development process and facilitating easier maintenance and future improvements.

Link:
https://www.canva.com/design/DAG0cZkrmUQ/QN10eJ9ir_v7urEMMcovng/edit?utm_content=DAG0cZkrmUQ&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton


CardioLink_Server
cardioLink.pojos
DiagnosisFile
PATIENT
DOCTOR
USER
cardioLink.enum
ROLE
SEX
cardioLink.jdbcInterfaces
Doctor Manager
Doctor getDoctorbyUserId(int userId) throws SQLException;
Patient getPatientByDiganosisFileID(int idDiagnosisFile)throws SQLException;
void downloadFileInComputer(DiagnosisFile diagnosisFile) throws IOException;
List<DiagnosisFile> listDiagnosisFiles(TODO);
List<DiagnosisFile> getAllDiagnosisFilesFromPatient(int idPatient);
void UpDateDiagnosisFile(DiagnosisFile diagnosisfile)throws SQLException;
String getFragmentOfRecording(int idDiagnosisFile, int position)throws SQLException ;
List<Boolean> getSateOfFragmentsOfRecordingByID(int idDiagnosisFile)throws SQLException;
void addDoctor(Doctor doctor) throws SQLException;
List<String> getAllPatientsInsuranceNumberbyDoctor()throws SQLException;
List<String> getAllFragmentsOfRecording(int id_DiagnosisFile) throws SQLException;
DiagnosisFile getDiagnosisFileByID(int idDiagnosisFile) throws SQLException;
List<Integer> getSequencesOfRecording(int idDiagnosisFile)throws SQLException;
Patient Manager
void register(String username, String password, String role);
void registerDoctor(String username, String password, String role);
//boolean verifyPassword(String username, String password);
String getRole(String username);
User getUserByCredentials(String username, String password) throws SQLException;
String getPassword(String username);
int getUserId(String username);
User getUserByUsername(String username);
Medical Record Manager
void addPatient(Patient p) throws SQLException;
Patient getPatientById(int idPatient) throws SQLException;
Patient getPatientByUserId(int userId) throws SQLException;
List<DiagnosisFile> getAllDiagnosisFilesFromPatient(int idPatient)throws SQLException;
String getFragmentOfRecording(int idDiagnosisFile, int position)throws SQLException;
List<Boolean> getSateOfFragmentsOfRecordingByID(int idDiagnosisFile)throws SQLException;
void addNewDiagnosisFile(DiagnosisFile diagnosisFile)throws SQLException;
int returnIdOfLastDiagnosisFile()throws SQLException;
void saveFragmentOfRecording(int idDiagnosisFile, String fragmentData,int sequence)throws SQLException;
int getNextSequenceNumber(Connection c, int idDiagnosisFile);
void updateSymptomsInDiagnosisFile(int idDiagnosisFile, String selectedSymptoms);
String getDoctornameByPatient(Patient loggedPatient);
List<String> getAllFragmentsOfRecording(int id_DiagnosisFile) throws SQLException;
jdbc
JDBCDoctorManager
JDBCPatientManager
JDBCUserManager
ConnectionManager
Encryption
thread
ServerClientThread
ServerAdminThread
PatientThread
ServerThread
PatientThread


CardioLink_Patient
cardioLink.jdbcInterfaces
BitalinoManager
BITALINO
DeviseDiscoverer
Frame
Executable
DoctorApplicationGUI (JFrame)
HomePanel
AuthPanel
LoginPanel
RegisterPanel
BitalinoRecordingPanel
SymptomsSelectorPanel
BitalinoRecordingPanel
cardioLink.pojos
DiagnosisFile
PATIENT
DOCTOR
USER
cardioLink.enum
ROLE
SEX


CardioLink_Doctor
Executable
DoctorApplicationGUI
DoctorMenuPanel
AuthPanel
RegisterPanel
ViewPatientPanel
LoginPanel
ViewDiagnosisFilePanel
CompleteDiagnosisFilePanel
ViewRecordingPanel
SearchPatientPanel
RecentlyFinishPanel
cardioLink.pojos
DiagnosisFile
PATIENT
DOCTOR
USER
cardioLink.enum
ROLE
SEX

7. Communication Implementation

7.1 Protocol Communication Test

The client's communication protocol was initially tested in the classes PatientServerConnection.java and DoctorServerConnection.java, it was with those two classes that we made sure our server worked for multiple clients simultaneously, however the implementation of the code for both those clients in the server has since changed. So it's possible that those two classes no longer do their intended functions.

We highly recommend only using the PatientSwing.java for the patient app, the Doctor.java for the Doctors and the AdminSwing.java for the administrator.

7.2 BITalino Communication Test

The bitalino recording was initially tested in the patient app. In the src/main/java/bitalino there is a class called TestBitalinoManager.java. That class did the initial testing of the bitalino recording and the class SignalFilePlotter.java plotted those signals in the files created when using the test. The signals recorded are stored in the patient app in a directory named firstRecordingsTXT.

The final communication from the bitalino to the server is done by a method called startRecordingToServer which starts recording ECG and EDA data for the specified patient. The data is sent in 10 second fragments to the server via OutputStreams.

8. Interface

In all three applications developed for this system, the Doctor, Patient, and Administrator interfaces, the graphical user interface was implemented using Java Swing. This framework was selected because it provides a robust, platform-independent set of components that allow us to build consistent, responsive, and easily maintainable interfaces across the different clients. Its event-driven architecture facilitated clear interaction handling between the user and the underlying network logic, ensuring reliable communication with the server in every module.

9. Troubleshooting

9.1 System Initialization and Connection Setup

Upon launching the application, the system validates several dependencies to ensure a stable environment. The following conditions must be met for a successful startup:

Server Side:

- Database Dependency: The server initialization is contingent upon a successful connection to the database. If the JDBC handshake fails, the server will not start.

Client Side:

- Socket Connectivity**:** Clients attempting to connect to an incorrect port will receive a connection refusal.
- Data Integrity: Conflicts regarding data input formats (e.g., mismatched data types sent to the server) are intercepted and handled internally by the application's exception management system, preventing application crashes.

Patient Client (Bitalino Integration):

- Device Pairing: The Patient client requires a valid Bluetooth connection to the Bitalino device. Failure to provide the correct MAC address or inability to establish a Bluetooth socket will prevent the recording module from initializing.

9.2 Runtime Exception Handling

The system includes handling mechanisms for interruptions occurring after the connection has been established:

Device Disconnection (Bitalino): In the event of a Bitalino Bluetooth disconnection during an active recording session, the application detects the signal loss and stops the recording process to prevent data corruption.

Server Connectivity Loss: If the TCP/IP connection to the server is severed unexpectedly (e.g., network failure), the client application catches the resulting IOException and notifies the user.

Doctor Client Concurrency: Communication with the Doctor client is maintained via a dedicated server-side thread. If this thread becomes unresponsive or terminates due to a connection error, the Doctor client is designed to handle the SocketException gracefully.