# SOFTWARE ARCHITECTURE

Coursework – Report

Matriculation Number: 40646460

# Contents

# Chapter 1: Architecture Selection for KwikMedical System

## 1.1 Introduction

The purpose of this chapter is to analyse the specifications of the KwikMedical system, then use this analysis to select two suitable architectural styles for its implementation. KwikMedical is a distributed system designed to handle critical healthcare operations, including patient call handling, ambulance crew management, and hospital data processing. Due to the distributed nature of the system, it requires an architecture that supports multiple clients interacting with a centralised service efficiently.

The three key quality metrics identified for KwikMedical are:

i. Security: To protect sensitive patient and operational data from unauthorised access.

ii. Scalability: To accommodate future growth, such as adding more clients or adding new features like Global Positioning System (GPS) tracking.

iii. Maintainability: To ensure that the system can be easily updated, debugged, and adapted over time.

## 1.2 Candidate Architectural Styles

Based on KwikMedical's requirements, several architectural styles were considered, but two candidates stood out as optimal: Client/Server Architecture and Three-Tiered Architecture. These will be analysed in detail in the following sections, each candidate is described and evaluated below in terms of its characteristics, benefits, drawbacks, and suitability for KwikMedical.

## 1.3 Candidate 1: Client/Server Architecture

The Client/Server Architecture is a distributed system design where multiple clients interact with a centralised server. The clients issue requests to the server which processes these requests, performs operations such as retrieving or updating data, and sends a response back to the clients. This architecture is used for systems requiring central management and lightweight client applications.

### 1.3.1 Characteristics of Client/Server Architecture

Components:

i. Clients: The devices or applications (e.g., call operator's computer, ambulance crew's mobile phone, hospital staff's terminal) that send requests to the server. These clients handle user interaction, but depend on the server for processing and data management.

ii. Server: A centralised system that processes client requests, applies business logic, and manages data.

Connectors**:** Communication between clients and the server is facilitated through message requests sent over a network (e.g., HTTP requests or socket-based connections can serve as the protocol for exchanging messages).

Communication**:** The process involves a client initiating a request (e.g., sending patient details) to the server. The server processes the request, performs the necessary operations (e.g., querying a database or dispatching an ambulance), and returns the result to the client. This interaction follows a call-and-response pattern.

### 1.3.2 Advantages of Client/Server Architecture

- Centralised Management: Data and logic are managed on the server, making it easier to manage security, updates, and data consistency.
- Simplicity: The straightforward design of clients interacting with a single server makes this architecture relatively easy to understand and implement.
- Scalability: Adding more clients is straightforward, as no changes are required to the architecture itself.

### 1.3.3 Disadvantages of Client/Server Architecture

- Single Point of Failure: If the central server becomes unavailable due to hardware failure or an attack, the entire system becomes inoperable.
- Limited Separation of Concerns: All processing and data management are centralised on the server, thereby reducing modularity and making updates or maintenance more challenging.
- Disruption Risk: Any server modifications can impact all connected clients, thus increasing downtime risk during upgrades or changes.

### 1.3.4 Suitability of Client/Server Architecture for KwikMedical

The Client/Server Architecture is a viable choice for KwikMedical, as it supports the system's distributed nature by allowing multiple clients (call operators, ambulance crew, and hospital staff) to communicate with a centralised server. Conversely, the lack of separation of concerns may complicate future updates or maintenance, whilst the single-point-of-failure risk makes it less ideal for a mission-critical healthcare system, as any downtime could compromise patient safety.

## 1.4 Candidate 2: Three-Tiered Architecture

The Three-Tiered Architecture is a layered system that divides functionality into three distinct layers:

i. Presentation Layer: Handles user interactions (e.g., entering data or displaying information).
ii. Business Logic Layer: Processes requests, applies system rules, and acts as an intermediary between the presentation and database layers.
iii. Database Layer: Stores and retrieves data needed for the application.

This architecture ensures a clear separation of concerns, with each layer being responsible for specific technical tasks. The separation improves scalability, maintainability, and security.

### 1.4.1 Characteristics of Three-Tiered Architecture

Components:

i. Presentation Layer: This is the user-facing client (e.g., the call operator’s application, ambulance crew’s mobile interface, and hospital staff’s system) which collects user input, and displays responses.

ii. Business Logic Layer: The server that processes requests from the clients, applies business logic (e.g., determining ambulance availability), and manages communication between the presentation and database layers.

iii. Database Layer: The structured query language (SQL) database that stores all critical data, including patient records, ambulance availability, and hospital resources.

Connectors: The layers communicate using Hypertext Transfer Protocol (HTTP) requests, Application Programming Interfaced (APIs), or database queries. An example could be: the presentation layer sends a request to the business logic layer which then queries the database layer and returns the result.

Communication:

- The presentation layer sends a user request (e.g., "Fetch patient details").

- The business logic layer processes this request, retrieves the necessary data from the database, and applies any rules (e.g., security checks).
- The database layer executes the query, then sends the data back to the business logic layer, which then forwards the response to the presentation layer.

This communication happens asynchronously to ensure modularity and efficiency.

### 1.4.2 Advantages of Three-Tiered Architecture

- Separation of Concerns: Each layer has a defined responsibility, making the system modular and easier to maintain.
- Scalability: Changes to one layer (e.g., updating the user interface or database) do not affect other layers, allowing for independent development and scaling.
- Enhanced Security: The business logic layer acts as a gatekeeper, ensuring that clients cannot directly access the database.
- RESTful API Integration: One of the key advantages of the Three-Tiered Architecture is its compatibility with modern web technologies, particularly RESTful APIs. The communication between the layers can be efficiently managed through RESTful APIs, which use standard HTTP methods (GET, POST, PUT, DELETE) to perform operations. RESTful APIs offer simplicity, scalability, and ease of integration, making them ideal for a distributed system like KwikMedical. They enable different clients (e.g., web applications, mobile apps, and external services) to interact seamlessly with the business logic layer, and access the system's functionalities in a standardised manner.

### 1.4.3 Disadvantages of Three-Tiered Architecture

- Latency: Communication between layers can introduce delays, especially if the system is not optimised.
- Development Complexity: Dividing responsibilities across layers can complicate the design and debugging process.
- Potential Bottlenecks: If the business logic layer is not designed well, it may become a processing bottleneck as the system scales.

### 1.4.4 Suitability of Three-Tiered Architecture for KwikMedical

The Three-Tiered Architecture is highly suitable for KwikMedical. By separating the user-facing interface, processing logic, and data storage, the system becomes more secure, scalable, and maintainable. For instance, adding new clients (e.g., a mobile application for the ambulance crew) would only require updates to the presentation layer. Additionally, the business logic layer ensures that sensitive medical data is protected by restricting direct access to the database. While the complexity of this architecture poses challenges, the benefits outweigh the drawbacks for a critical healthcare system like KwikMedical.

## 1.5 Comparison of Candidate Architectures

### 1.5.1 Summary of Candidate Evaluation

The two architecture styles evaluated for KwikMedical (i.e., Client/Server and Three-Tiered) each possess distinct strengths and weaknesses. The Client/Server model emphasises simplicity and ease of implementation, making it suitable for small systems with centralised control; however, it struggles with scalability, maintainability, and security, especially in distributed environments like KwikMedical. Conversely, the Three-Tiered Architecture, with its separation of presentation, business logic, and data storage, offers better modularity, security, and scalability, albeit at the cost of higher complexity and potential latency.

### 1.5.2 Final Architecture Selection

Based on the requirements of KwikMedical (including: scalability, security, and maintainability), the Three-Tiered Architecture is the preferred choice. The reasons for this are outlined below:

i. It aligns with the need for modular development, allowing for independent management of each layer.
ii. Its robust security framework ensures that sensitive medical data is well-protected.
iii. Its scalability supports the system's growth to accommodate increasing demands from clients like call operators, hospitals, and ambulances.

## 1.6 Expected Quality of Chosen Architecture

### 1.6.1 Scalability

The Three-Tiered Architecture is inherently scalable as each layer (i.e., presentation, business logic, and data storage) can be independently scaled to handle increased load. For instance, during high demand (e.g., emergencies) the system can deploy additional servers for the presentation layer to handle user interactions, or expand the database layer to manage increased data traffic. This modular scaling ensures that performance is maintained without overhauling the entire system. Additionally, this architecture supports integrating cloud services, thereby enhancing elastic scalability to adapt to varying workloads.

### 1.6.2 Security

The security benefits of the Three-Tiered model stem from its compartmentalisation. Direct database access is restricted, as interactions go through the business logic layer, reducing vulnerabilities. Additionally, individual layers can implement dedicated security measures (e.g., encrypted data storage in the database layer and API gateways in the business logic layer) to handle authentication and prevent unauthorised access. The use of protocols such as Transport Layer Security (TLS) for secure communication between layers, ensures robust protection against threats like data breaches or unauthorised intrusions.

### 1.6.3 Maintainability

The modular design of the Three-Tiered Architecture significantly improves maintainability. Changes to one layer (e.g., updating the user interface in the presentation layer or upgrading database technology) can be made without disrupting the other layers. This separation of concerns also simplifies debugging and testing, as issues are isolated to specific layers. Development teams can work on different layers simultaneously, thus enhancing productivity and reducing downtime during updates or system enhancements.

In conclusion, the Three-Tiered Architecture's focus on modularity, security, and scalability positions it as the ideal architecture for a critical healthcare system like KwikMedical, ensuring long-term performance, reliability, and adaptability.

# Chapter 2: System Design

## 2.1 Introduction

This chapter provides an overview of the system design (prototype), detailing its architectural layout and individual components. The system will use a Three-Tiered Architecture to ensure separation of concerns, scalability, and maintainability. The three tiers include the Presentation Layer, implemented with a WPF (Windows Presentation Foundation) application, the Business Logic Layer, developed using an ASP.NET Web API, and the Database Layer, powered by Microsoft SQL Server Express.

## 2.2 System Architecture Overview

The system architecture follows a three-tier design pattern:

- Presentation Tier (Client Tier): This layer includes a WPF application that serves as the graphical user interface (GUI) for the end-user. It handles user interaction and displays data retrieved from the backend.
- Business Logic Tier (Server Tier): The ASP.NET Web API acts as the intermediary between the presentation layer and the database. It encapsulates the business rules and logic while ensuring secure data access.
- Database Tier (SQL Server): A relational database implemented with Microsoft SQL Server (express), which stores the application's persistent data. The Web API directly communicates with this tier for data storage and retrieval. In this prototype build, server configuration is locally hosted, meaning any new machine must configure SQL Server in the appsettings.json file inside the ASP.NET Web API Project.

These tiers interact using HTTP/HTTPS protocols, and EntityFramework, ensuring decoupled communication and modular functionality.

## 2.3 Detailed Design of Each Tier

### 2.3.1 Presentation Tier (WPF Application)

The Presentation Layer is implemented using a WPF application, providing distinct interfaces for different system users. Each interface is tailored to specific roles and responsibilities, ensuring that the application is user-friendly and efficient for its intended audience.

Call Operator Window (See Appendix 1)

Purpose: Designed for call operators managing emergency calls. The window provides features for:

- Entering caller details, and location data.
- Assigning appropriate response units (e.g., ambulances) to the reported incidents.

Components:

- Textboxes for data entry.
- Buttons for saving call records.
- Button to send patient record to the active ambulance crew.
- Button to clear the current record.
- Search box to search for an existing record by CHI number.

Communication: Sends call data as HTTP POST requests to the ASP.NET Web API/calls data as HTTP GET(by ID) requests to the ASP.NET Web API.

Ambulance Phone Window – **Phone Simulation** (See appendix 2)

Purpose: Simulates a mobile phone interface for ambulance crews. This interface allows paramedics to:

- Receive details about assigned emergency calls.
- Update case progress in real-time.

Components:

- Incident details form displayed in a concise format.
- Patient details displayed.
- Submit response button to store the response on KwikMedical.

Communication: Sends HTTP PUT requests to update the incident status in the backend.

Regional Hospital Window (See appendix 3)

Purpose: Used by hospital personnel to track incoming patients.

- Displays incoming ambulance cases and estimated arrival times.

Components:

- Dashboard with incident summaries and patient details.

Communication: Pulls real-time case data from the Web API.

### 2.3.2 Business Logic Tier (ASP.NET Web API)

In this section, a discussion of the structure of the Business Logic Layer of the application, which is implemented through ASP.NET API. The Business Logic Layer (BLL) is the critical intermediary between the presentation layer (WPF) and the database layer (SQL Server), containing services, repositories, and controllers that handle all application-specific logic and data access.

The following components play an essential role in ensuring the smooth functioning of the business logic layer:

Controllers:

The Controllers in ASP.NET API are responsible for processing incoming HTTP requests, interacting with the services, and returning appropriate responses. Controllers define the endpoints that clients (in this case, the WPF application) use to interact with the system.

For instance, there are two primary controllers used in the system:

i. AmbulanceRecordsController: Manages operations related to ambulance records. It includes methods for creating new records (POST /api/ambulancerecords) and fetching ambulance records by their ID (GET /api/ambulancerecords/{id}).
ii. PatientsController: This controller handles the patient-related operations, such as getting all patients, retrieving a specific patient by their CHI (Community Health Index) number, and creating new patient records.

These controllers interact with the services to perform business logic operations (e.g., validation, transformations, and data manipulation) before interacting with the database or responding to the client

Models

The Models define the structure of the data that is passed between the application layers. These are C# classes that represent real-world entities, such as AmbulanceRecord and Patientcs.

- AmbulanceRecord: This model represents the ambulance record, capturing essential details such as the action taken, response time, and location.

- Patientcs: This model represents patient data, including the patient's name, CHI number, medical conditions, and address.

Models help ensure data consistency and are used to validate the data being passed through the application.

## Repositories

Repositories abstract the interaction with the database, providing methods for data retrieval, insertion, updating, and deletion. They offer a cleaner interface between the business logic and the database, encapsulating SQL queries or ORM calls in a way that can be reused across different parts of the application.

- PatientRepository: This repository provides methods to interact with patient data (e.g., such as getting all patient records, finding a patient by ID or CHI number, and adding, updating, or deleting a patient record).
- AmbulanceRecordRepository: Similar to the PatientRepository, this repository handles operations related to AmbulanceRecord data. It offers functionality for fetching records, adding new ones, and saving changes to the database.

The repository pattern helps to decouple the data access logic from the rest of the application and facilitates easier unit testing and maintenance.

## Services

The Service Layer contains the application's core business logic. It acts as an intermediary between the repositories and the controllers, processing data according to the application's business rules.

Examples:

- PatientService: Handles patient-related business logic, including retrieving patients, validating inputs, and interacting with the PatientRepository.
- AmbulanceRecordService: Manages the business logic for ambulance records, such as validating data and coordinating with the AmbulanceRecordRepository.

These services are invoked by the controllers to perform operations on the data before responding to client requests. They encapsulate business logic and ensure that the application operates as intended.

### 2.3.3 Database Tier (SQL Server Tier)

Purpose: Stores and manages the system’s data in a structured format.

Database Design:

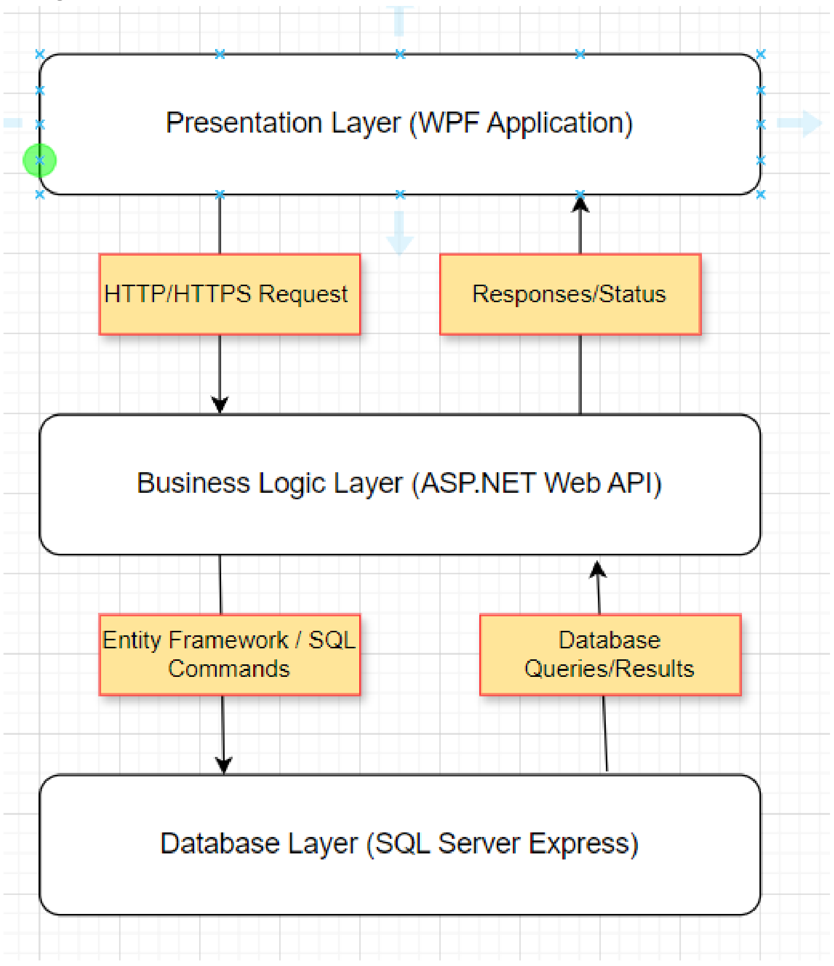
- Schemas: Define the structure of tables, relationships, and constraints.
- Stored Procedures and Queries: Optimized SQL queries and stored procedures for data manipulation.

## 2.4 Communication Between Layers

The layers interact as follows:

1. Presentation to Business Logic: The WPF application communicates with the ASP.NET Web API through HTTP/HTTPS requests. These requests may include JSON payloads containing user inputs or application data.
2. Business Logic to Database: The ASP.NET Web API uses Entity Framework or raw SQL queries to interact with the SQL Server, retrieving and persisting data as necessary.
3. Database to Business Logic: The SQL Server responds to queries with the requested data, which the Web API processes before sending it back to the client.
4. Business Logic to Presentation: The Web API responds to WPF requests with JSON data or appropriate status codes, ensuring a smooth flow of information.

Diagram:


Presentation Layer (WPF Application)
HTTP/HTTPS Request
Responses/Status
Business Logic Layer (ASP.NET Web API)
Entity Framework / SQL Commands
Database Queries/Results
Database Layer (SQL Server Express)

By adhering to the three-tiered design, the system ensures modularity, maintainability, and scalability, making it well-suited for future enhancements.

# Chapter 3: Evaluation

## 3.1 Evaluation Criteria

In evaluating the KwikMedical system's design and implementation, several key criteria will be considered. These criteria include:

- Performance: The system's ability to handle the expected workload, including the speed and efficiency of data retrieval and processing.
- Scalability: The system's ability to accommodate future growth, whether that means supporting more users, scaling to handle additional features, or adapting to larger datasets.
- Security: The robustness of the system's measures to protect sensitive healthcare data and prevent unauthorized access or breaches.

- Usability: The ease of use for various stakeholders (e.g., call operators, ambulance crews, and hospital staff), ensuring they can perform their tasks effectively.
- Maintainability: How easily the system can be updated, debugged, and extended over time, considering both the development and operational perspectives.

## 3.2 Strengths

The KwikMedical system, designed with a Three-Tiered Architecture, boasts several key strengths:

- Modularity: The Three-Tiered Architecture clearly separates the system into presentation, business logic, and database layers, making the system highly modular. This modularity ensures that different parts of the system can be developed, tested, and maintained independently, which simplifies debugging, updates, and scaling. The separation of concerns inherent in the Three-Tiered model provides multiple layers of security. E.g., the business logic layer acts as a gatekeeper to the database, ensuring that direct access is restricted. Moreover, security features like secure communication protocols (e.g., HTTPS) help safeguard sensitive medical data.
- Scalability: The system's design facilitates easy scaling. Each tier—presentation, business logic, and database—can be scaled independently. E.g., additional presentation servers can be deployed to handle increased client load, or additional functionality (e.g., GPS tracking) could be implemented smoothly.
- Maintainability: Separation of responsibilities between layers also contributes to better maintainability. Changes made to one layer (e.g., upgrading the user interface or adding new database features) can be implemented with minimal disruption to other layers .
- Efficiency: The system's HTTPS communication between layers ensures efficient, decoupled communication, enabling real-time data updates and a smoother user experience. The ASP.NET Web API in the business logic layer is well-suited for handling HTTP requests at scale .

## 3.3 Limitations

While the KwikMedical has several strengths, it also has a few limitations:

- Complexity: The Three-Tiered Architecture, while beneficial for modularity and scalability, can introduce complexity in development and maintenance. Dividing the system into three distinct layers requires careful coordination, and errors in one layer can cascade through to others. This could lead to higher initial development costs .

- Latency: interaction between layers, if not optimized, can introduce latency. E.g., every user request requires a round-trip through the presentation, business logic, and database layers, which might cause delays in response times, particularly during peak demand .
- Single Point of Failure (SPOF): While the Three-Tiered inherently supports scalability, a failure in the business logic layer or database layer can still cause significant disruptions. Without a robust failover mechanism, this could impact the system's reliability .
- Development Time: Due to the layered nature of the design, the development might take longer, as developers must ensure that all layers are well-integrated and functioning as intended. This leads to optimal hospital location functionalities not being implemented in this build.

## 3.4 Future Improvements

There are several potential improvements that could be made to the KwikMedical system:

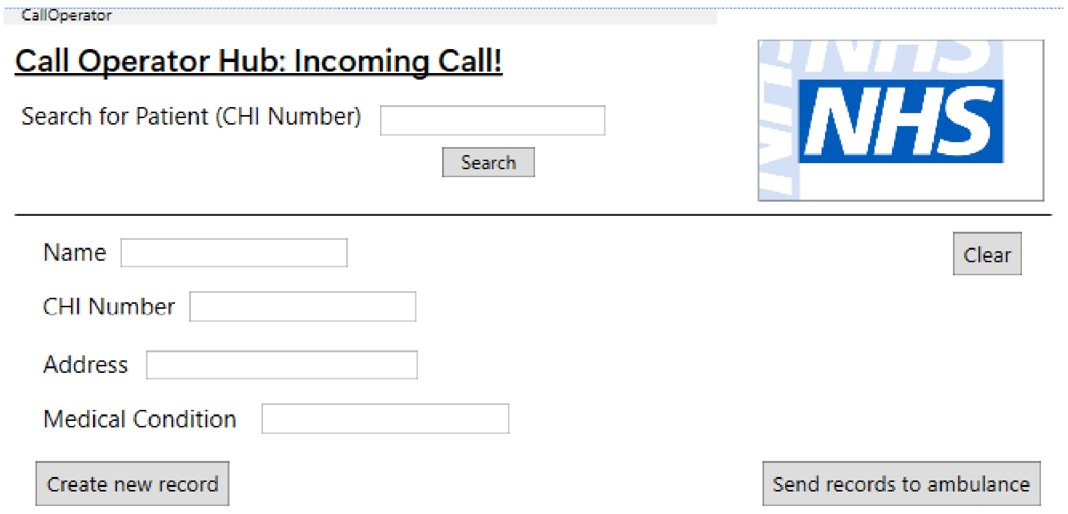
- Cloud Integration: Incorporating cloud infrastructure could enhance scalability even further. By leveraging cloud services, the KwikMedical server would not need to run locally, but instead be hosted via the cloud.
- Enhanced Security Features: Although the current system is designed with security in mind, future versions could include even more robust measures, like multi-factor authentication (MFA) for users accessing the system, or more advanced encryption methods to protect patient data both at rest and in transit .
- Integration into mobile applications: Future versions of KwikMedical could integrate the server into new mobile clients, based off IOS and Android programming.
- AI and Machine Learning: To improve decision-making, AI could be integrated into the system. E.g., predictive ML could help forecast patient needs based on historical data, enabling better resource allocation and reducing response times.

## 3.5 Conclusion

The KwikMedical system, built on the Three-Tiered Architecture, offers a robust and scalable solution for medical healthcare operations. Its strengths in modularity, security, and scalability make it well-suited for a distributed healthcare environment where performance and data integrity are paramount. However, the system also faces challenges related to complexity, latency, and potential single points of failure. By implementing future improvements such as cloud integration, the system can continue to evolve and meet the growing demands of the healthcare sector. Overall, the system is a solid foundation for a scalable, maintainable, and secure healthcare management solution.

# Appendices

## Appendix 1:



Call Operator Hub: Incoming Call!

Search for Patient (CHI Number)

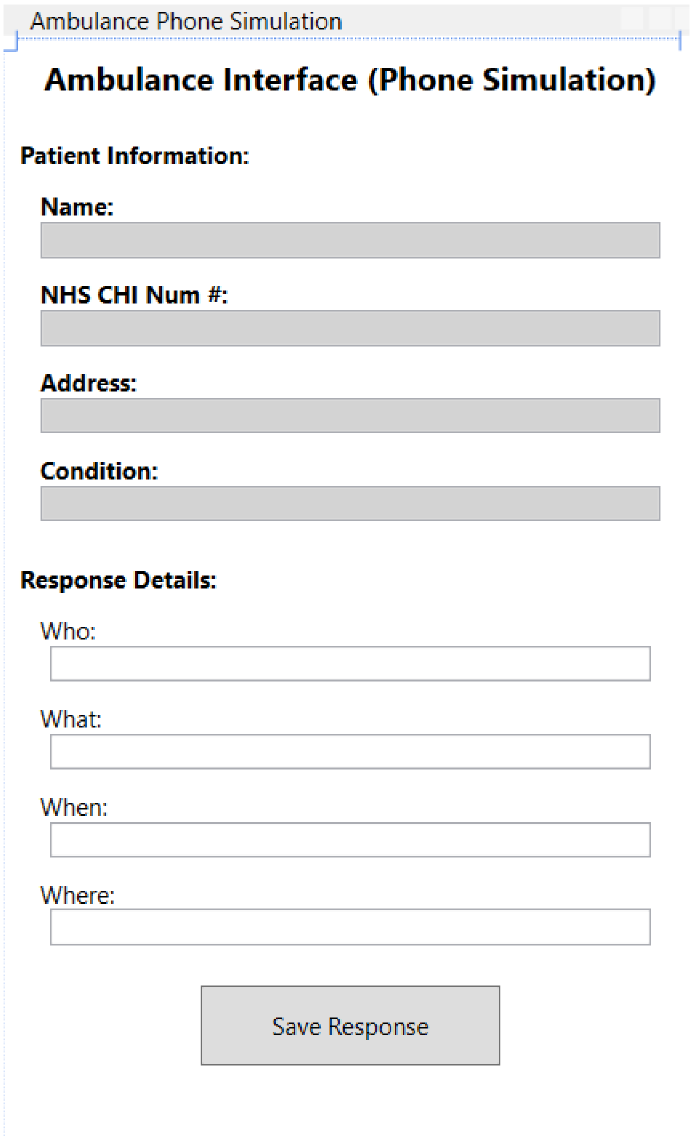
Name

CHI Number

Address

Medical Condition

Appendix 2:



Ambulance Phone Simulation

**Ambulance Interface (Phone Simulation)**

**Patient Information:**

**Name:**

**NHS CHI Num #:**

**Address:**

**Condition:**

**Response Details:**

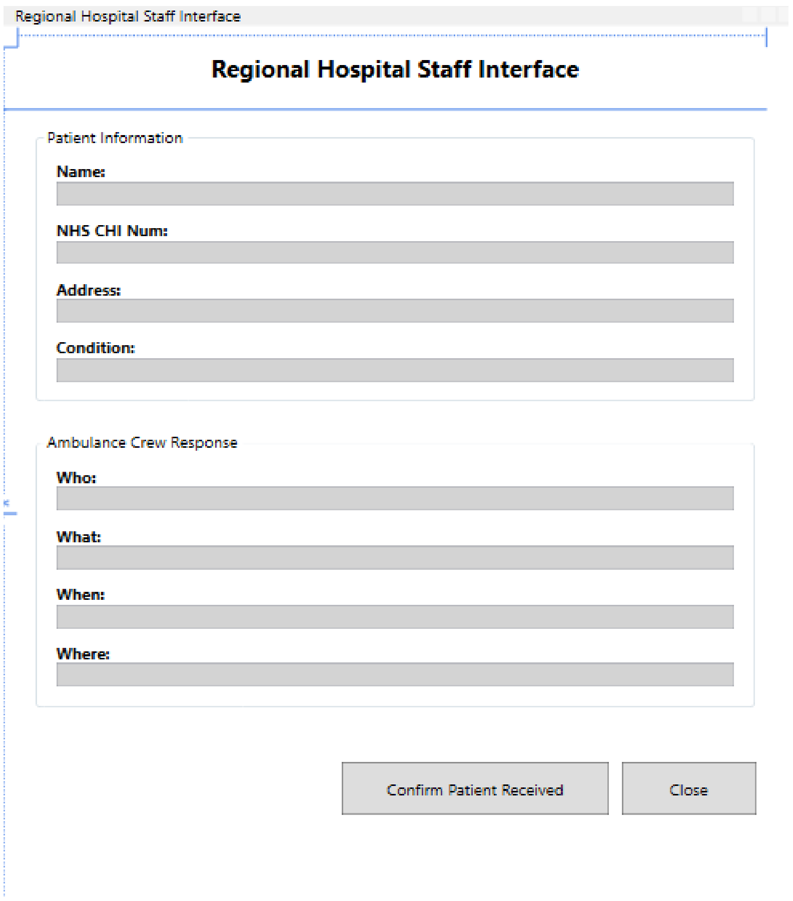
Who:

What:

When:

Where:

Appendix 3:


Regional Hospital Staff Interface

Regional Hospital Staff Interface

Patient Information

Name:

NHS CHI Num:

Address:

Condition:

Ambulance Crew Response

Who:

What:

When:

Where: