**Team 38 – Patient & Insurance Management System**

**Software Design**

**CSCI-P465/565 (Software Engineering I)**

**Project Team**

### Zane Ellis Snider

### Sri Rashmitha Boya

### Rishi Sanjaykumar Patel

### Harshitha Nooli

**Sahithi Vasireddy (Client)**

## 1. Introduction

Our design philosophy is rooted in the belief that a well-designed healthcare management system should prioritize user-friendliness, security, scalability, and maintainability. Our primary design goals include:

* Creating an intuitive and user-friendly interface for patients, doctors, and insurance providers.
* Optimizing system performance to deliver a responsive and efficient user experience.

### 1.1 System Description

The Patient and Health Insurance Management System is a comprehensive software solution designed to address the evolving needs of patients, healthcare providers, and insurance providers within the healthcare industry. This system aims to streamline the management of health insurance information, medical records, and appointment scheduling, providing a unified platform for effective healthcare administration.

### 1.2 Design Evolution

The design evolution of our Patient and Health Insurance Management System reflects a user-cantered approach, guided by stakeholder feedback. Our architectural choices prioritize scalability, security, and maintainability. The selection of technologies, security measures, and UI design align with our core project objectives, ensuring a robust and user-friendly solution.

#### 1.2.1 Design Issues

The design of our Patient and Health Insurance Management System is influenced by key considerations, including the need for a web-based platform with cross-browser compatibility for widespread accessibility. We prioritize RESTful API protocols for seamless data exchange and emphasize secure remote access. Stringent security measures, scalability for future growth, third-party integrations, accessibility standards, and user experience are driving factors. Comprehensive documentation and collaborative tools support efficient development and system maintenance, ensuring it effectively meets the complex demands of the healthcare domain.

#### 1.2.2 Candidate Design Solutions

In our project's early stages, we deliberated on several candidate design solutions to determine the most suitable approach for our Patient and Health Insurance Management System. These options included a monolithic architecture, which simplifies development but may pose scalability and maintenance challenges as the system grows. Alternatively, a microservices architecture was explored, offering flexibility and scalability, but introducing complexities in service management. We scrutinized various database management systems, such as PostgreSQL and MongoDB, to assess their compatibility with our data storage needs. For user authentication, methods like OAuth and token-based authentication were considered, each evaluated for its ability to ensure robust security. Additionally, we compared UI frameworks like React.js and Angular to optimize frontend development. Our team also explored cloud hosting providers, including AWS, Azure, and Google Cloud, to identify the most suitable platform for deployment. Finally, we examined the feasibility of integrating external services, such as mapping and machine learning, to enhance the system's capabilities. Ultimately, our selected design choices align with our project's primary objectives, striking a balance between efficiency, scalability, security, and user experience.

#### 1.2.3 Design Solution Rationale

The selected design option for our Patient and Health Insurance Management System is based on a careful analysis of its ability to meet the project requirements effectively and to address the design issues identified. This choice reflects our commitment to creating a robust, scalable, and user-friendly healthcare management platform.

**PostgreSQL Database:** Our decision to use PostgreSQL as the database management system is rooted in its proven ability to handle complex relational data efficiently. This aligns with our requirement for data integrity and comprehensive healthcare record management. PostgreSQL's extensibility and support for advanced features also make it a suitable choice for accommodating future expansion.

**OAuth for User Authentication:** OAuth was selected as the primary method for user authentication due to its industry-wide adoption and robust security features. This choice ensures that user access is secure, with strict authorization controls in place. It addresses our design issue of stringent security requirements and compliance with data protection regulations.

### 1.3 Design Approach

#### 1.3.1 Methods

The project follows a component-based software development model with Agile project management. The component-based approach allows for code-reuse and extensibility compared to other methods. Since the project has similar features for various components like patient, doctor and insurance providers, this decision was chosen.

#### 1.3.2 Standards

Strict code lint analysis tools integrated into the IDEs (Integrated Development Environments) used by the team members ensure that the code quality is maintained throughout the application. This manifests in the form of following conventions like CamelCase, meaningful naming conventions, etc.

#### 1.3.3 Tools

Some tools we intend to use for the development of the application are:

1. Visual Studio Code (for software development)
2. JIRA (for project management)
3. Tools like Rational Rose/Lucid App for UML diagrams and system design documents.

## 2. System Architecture

### 2.1 System Design

### 

### The above diagram depicts the high level system design of the application. It consists of a front-end component (written in REACT.JS), a back-end component (written in Python-Flask) and a database (PostgreSQL).

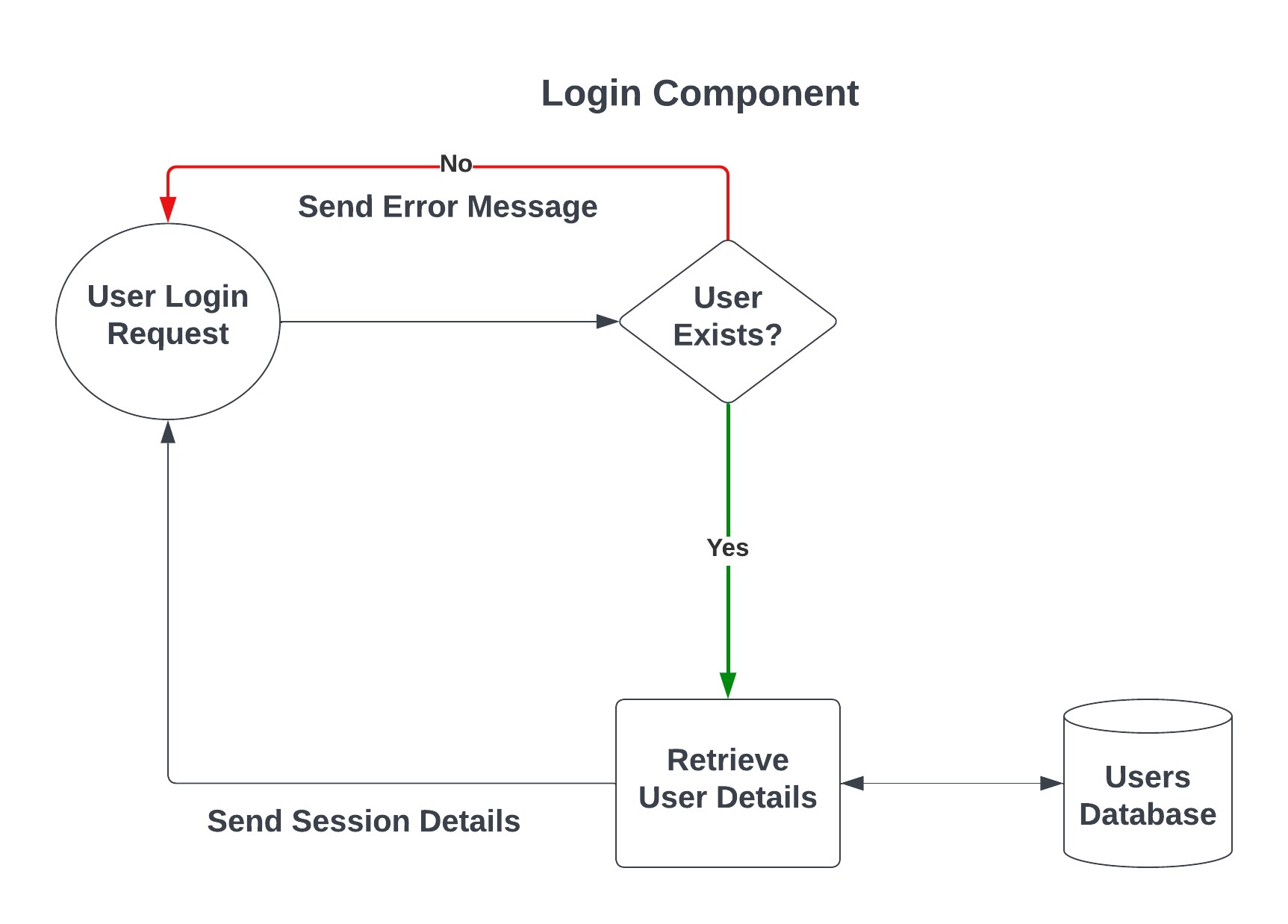
### 2.2 External Interfaces

* Payment Gateways: In order to handle consumer payments for insurance premiums and claims, the system will need to communicate with payment gateways.
* Medical Databases: In order to verify consumer claims, the system will need to communicate with medical databases to retrieve details about medical procedures and treatments.
* Insurance Providers: In order to verify customer policies and claims, the system will need to communicate with insurance providers.

## 3. Component Design The Component Design section details the proposed design of each system component. A system component is a functional partition of the system. Components may be organized as you see fit - a component may be a collection of objects, or a single object. However, a system must be composed of multiple components (that is, a system cannot be one component). The layout of this section is at your discretion, but please include the following (at a minimum) information for each component:

## Login Module

* **Description:** The Login module deals with logging in users into the application using their credentials like username/email and password. It also includes data type checking and validation for username and password.
* **Team:**  Zane Ellis Snider (primary - backend), Rishi Sanjaykumar Patel (secondary - backend), Harshitha Nooli (frontend), Sri Rashmitha Boya (Database)
* **Component Diagram:**



* **Component User Interface:** The user interface consists of a React.js form with required fields like username and password. It is initially designed in an interactive and visually pleasing interface written in React.JS.
* **Component Objects:**

The various components in the interface include:

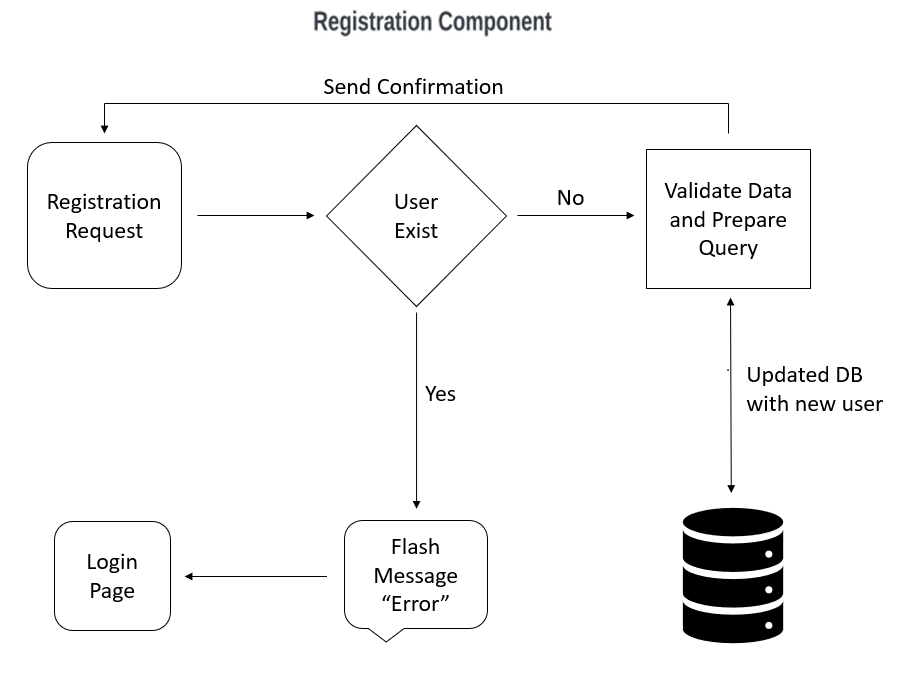
* React.js on the front-end.
* Error messages and error handling incase of any retrieval issues.
* **Component Interfaces (internal and external):**

The component interfaces with the following:

* Frontend: REACT.JS form which acts as the collector for user information.
* Backend in the form of a PostgreSQL request that is handled by the Python-Flask application running on the server. This module parses the incoming data, prepares it in the form of a query.
* Database: Indirect interface via the backend query run by Python. Retrieves user details based on the given input data. May return user details if the user exists or NULL if they don’t.
* **Component Error Handling:**
* Basic user input data type handling performed by the JS form. For example: only properly formatted user emails are accepted and only valid format/text passwords are allowed.
* Python-Flask prevents unauthorized data access by limiting the request method only to POST. All other requests are outright rejected. Also uses proper HTTP return codes for invalid/improperly formatted requests.
* Database enforces error handling by utilizing Python’s inbuilt ORM system for building the query and not relying on the user to define the query manually.

## Register Module

* **Description:** The Registration module deals with registering and adding users into the application using their credentials like username/email, password and other details like name. It also includes data type checking and validation for username and password.
* **Team:**  Zane Ellis Snider (primary - backend), Rishi Sanjaykumar Patel (secondary - backend), Harshitha Nooli (frontend), Sri Rashmitha Boya (Database)
* **Component Diagram:**



* **Component User Interface:** The user interface consists of a React.js form with required fields like username and password. It is initially designed in an interactive and visually pleasing interface written in React.JS.
* **Component Objects:**

The various components in the interface include:

* React.js on the front-end.
* Error messages and error handling incase of any retrieval issues.
* **Component Interfaces (internal and external):**

The component interfaces with the following:

* Frontend: REACT.JS form which acts as the collector for user information.
* Backend in the form of a PostgreSQL request that is handled by the Python-Flask application running on the server. This module parses the incoming data, prepares it in the form of a query.
* Database: Indirect interface via the backend query run by Python. Retrieves user details based on the given input data. May return user details if the user exists or NULL if they don’t.
* **Component Error Handling:**
* Basic user input data type handling performed by the JS form. For example: only properly formatted user emails are accepted and only valid format/text passwords are allowed.
* Python-Flask prevents unauthorized data access by limiting the request method only to POST. All other requests are outright rejected. Also uses proper HTTP return codes for invalid/improperly formatted requests.
* Database enforces error handling by utilizing Python’s inbuilt ORM system for building the query and not relying on the user to define the query manually.

**Revision History**

|  |  |  |
| --- | --- | --- |
| **Revision** | **Date** | **Change Description** |
| Initial Version | 09/29/2023 | Inception of Document for Sprint 1 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |