Power Rangers

AI Copilot Application
Software Architecture Document

Version 1.0

Revision History

Date	Version	Description	Author
23/11/24	1.0	First version of software architecture document	Vũ Thanh Việt
04/12/24	1.0	Add Deployment and Implement view	Trần Lê Bảo Duy

Table of Contents

Contents

Software Architecture Document

1. Introduction

1.1. Purpose

The Software Architecture Document provides an overview of the AI copilot application architecture design that has been implemented in the system. This document also illustrates the operations and components involved in the system architecture.

1.2. Scope

The Software Architecture Document covers the high-level structure of the system, including major components and interactions between them. The structure is designed based on the description of the system given in the Vision Document and Use-case specification Document.

1.3. Definitions

AI Chat PR: is an multi-AI-driven web-based platform designed to assist users in streamlining workflows, enhancing productivity, and automating repetitive tasks.

Customers: represent employees from various companies who use the AI Copilot application or people from 18-55 years old. They contribute by enriching the agents with domain-specific knowledge and using the agents' capabilities to efficiently accomplish tasks tailored to their organization's needs.

Admin: the team that manages both customers and the website's system and has the responsibility of operating the system.

MVC: https://viblo.asia/p/tat-tan-tat-ve-mo-hinh-mvc-Rk74avjAJeO

1.4. Acronyms, abbreviations

None.

1.5. References

- Software Engineering, 9th Edition, by Ian Sommerville, Addison Wesley, 2011
- Lecture slides by teacher Nguyen Minh Huy.

1.6. Overview

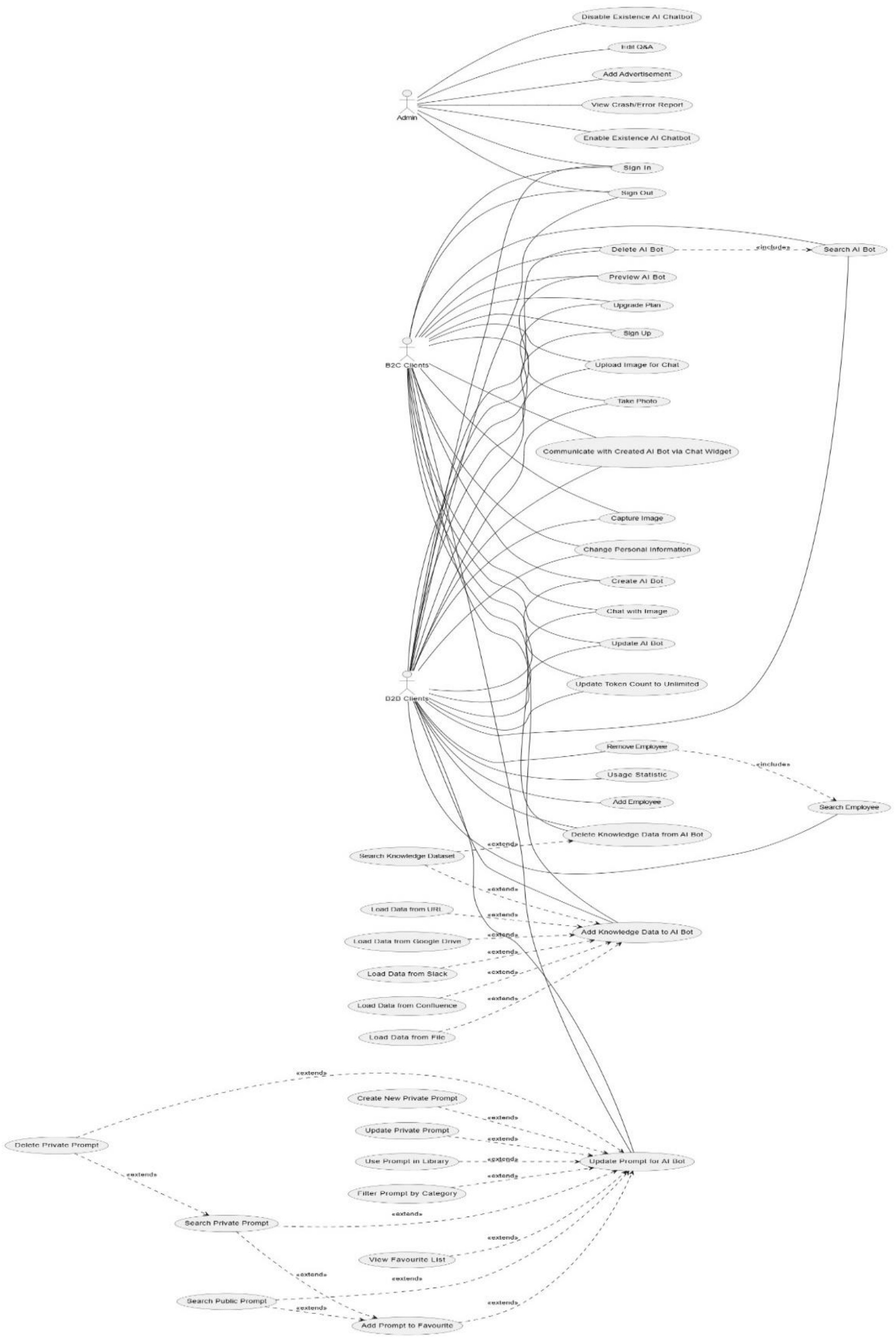
- Instruction
- Architectural Goals and Constraints
- Use-Case Model
- Logical View
- Development
- Implementation view

2. Architectural Goals and Constraints

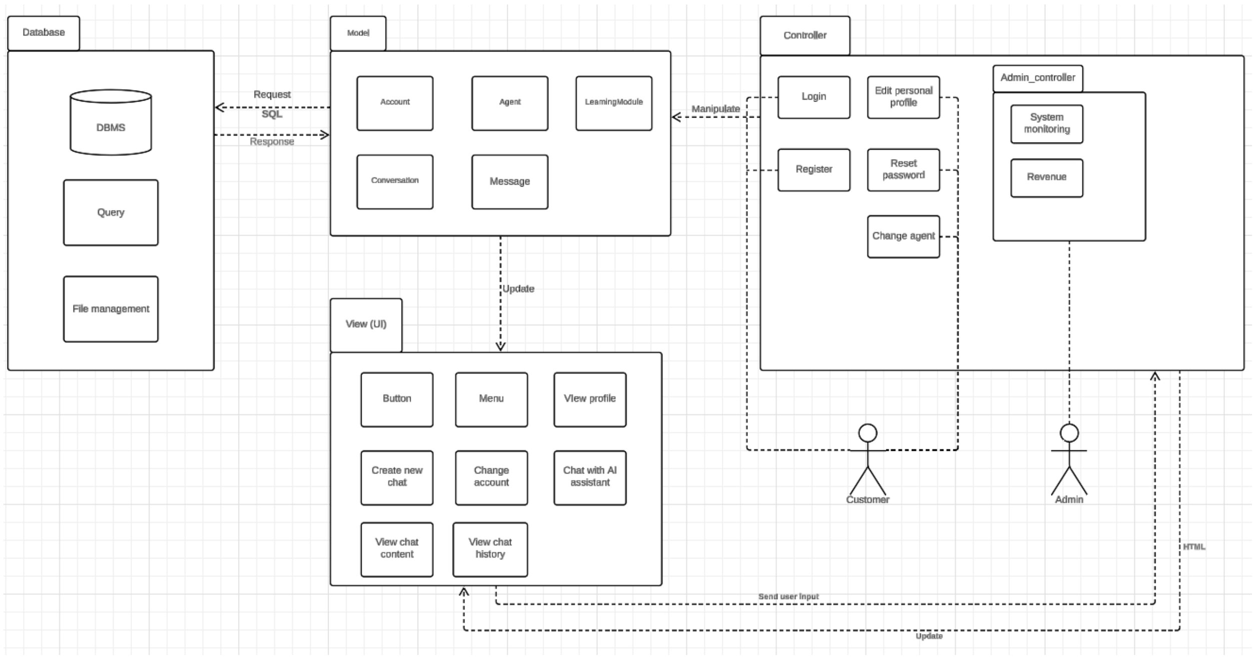
- **Runtime enviroment:** NodeJS
- **Framework:** Figma
- **Standards and Platform Requirements**
  - **Browsers:** Latest stable versions of popular web browsers.
  - **Mobile Platforms:** Compatibility with iOS and Android.
- **Performance Requirements:**

  - **Response Time:** Load operations within 2 seconds for most users.
  - **Concurrent Users:** Support at least 200 concurrent users during peak hours.
  - **Data Retrieval:** Execute queries within 500 milliseconds.
- **Robustness and Fault Tolerance:**
  - **Fault Tolerance**: Implement automatic failover and backup.
  - **Error Recovery**: Recover gracefully from errors and provide clear instructions.
  - **Data Backup**: Regular user data backups to prevent loss.

3. Use-Case Model



4. Logical View



The system is structured following the **Model-View-Controller (MVC)** architecture, as depicted in the diagram. Each component in the architecture plays a distinct role in handling data, managing user interactions, and rendering the user interface. Below is a description of the system based on the provided diagram.

1. Model: Data Management

The **Model** is the core of the system's data layer, responsible for managing and processing data. It communicates directly with the **Database** to retrieve or store data, ensuring data consistency and integrity. Key elements in the Model include:

- **Account**: Handles user account details and authentication.
- **Agent**: Represents the AI assistant or chatbot.
- **Learning Module**: Contains learning materials or training data for agents.
- **Conversation**: Manages chat histories and ongoing user interactions.
- **Message**: Handles individual messages exchanged in conversations.

The Model interacts with the **Database** through **Queries**, with data being passed back and forth using **SQL**.

2. View (UI): Present data to user

The **View** is responsible for generating the user interface (UI) and presenting data to the system's users in a visually intuitive way. It consists of several UI components:

- **Buttons** and **Menus**: Allow users to navigate and interact with the system.
- **Create new chat**: Enables initiating new conversations.
- **View chat history**: Displays previous conversations.
- **View profile**: Shows user account details.

- **Chat with AI assistant**: communication with the AI agent.

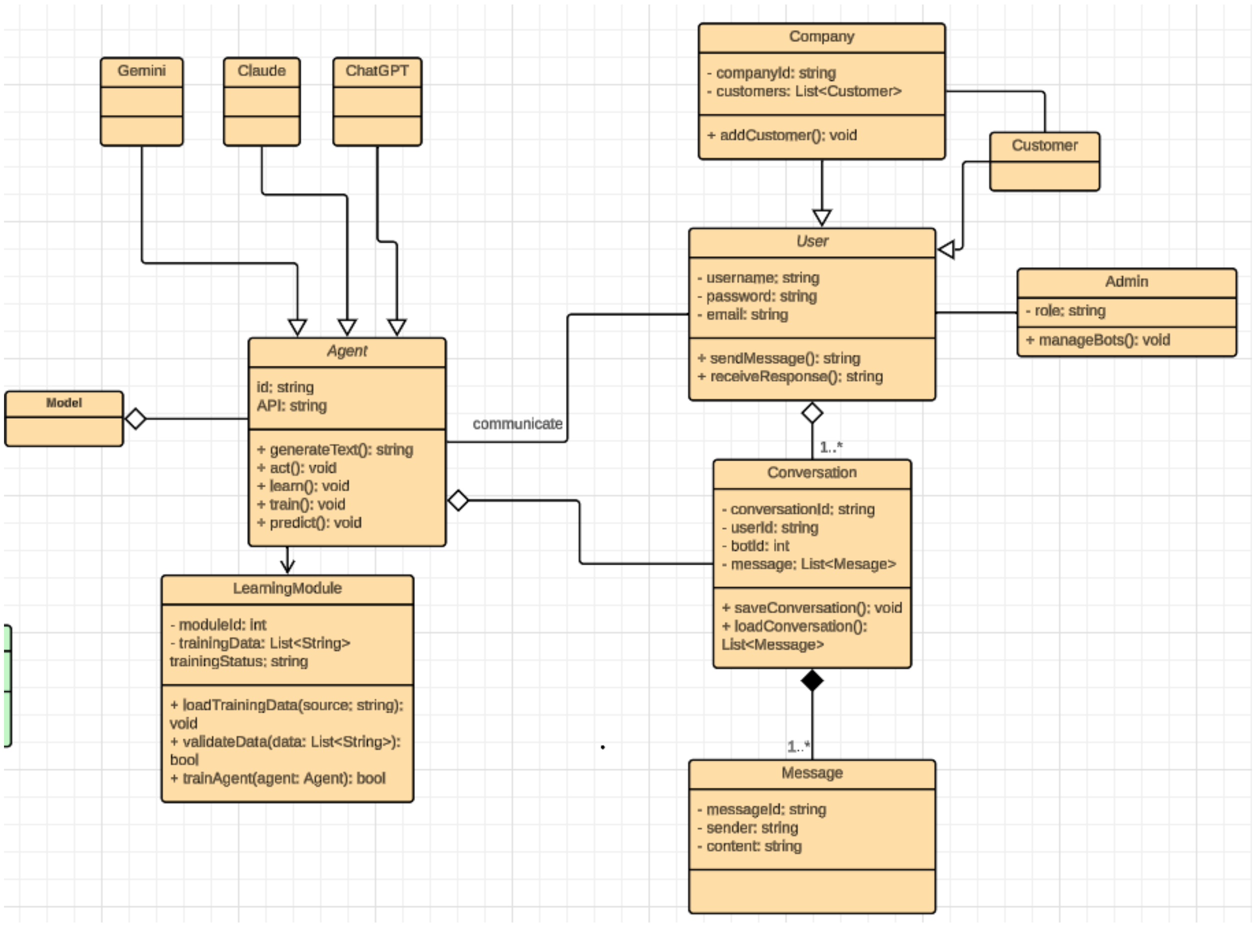
The View is updated dynamically based on input or updates received from the **Controller**.

3. Controller: Handling User Interactions and Updates

The **Controller** acts as the intermediary between the **Model** and the **View**, processing user input and updating the system as necessary.

The **Controller** ensures that data flows seamlessly between the **View** and the **Model** and applies business logic where required.

4.1 Component: Model



Agent

- Responsible for representing chatbot agents that interact with users and perform specific actions.
- Includes methods for text generation, learning, training, and prediction.
- Parent class for Gemini, Claude, and ChatGPT.

LearningModule

- Responsible for handling training data and processes to support agent development.

- Includes methods for loading data, validating data, and training agents.

User

- Responsible for setting up user objects within the structure.
- Used to manage users and enable interactions with the application.
- Parent class for Customer and Admin.

Customer

- Inherits from User.
- Represents system customers, with properties like active status and shopping-related data.

Admin

- Inherits from User.
- Represents administrative users responsible for managing system bots.

Company

- Responsible for managing customers associated with a specific company.
- Includes methods for adding customers.

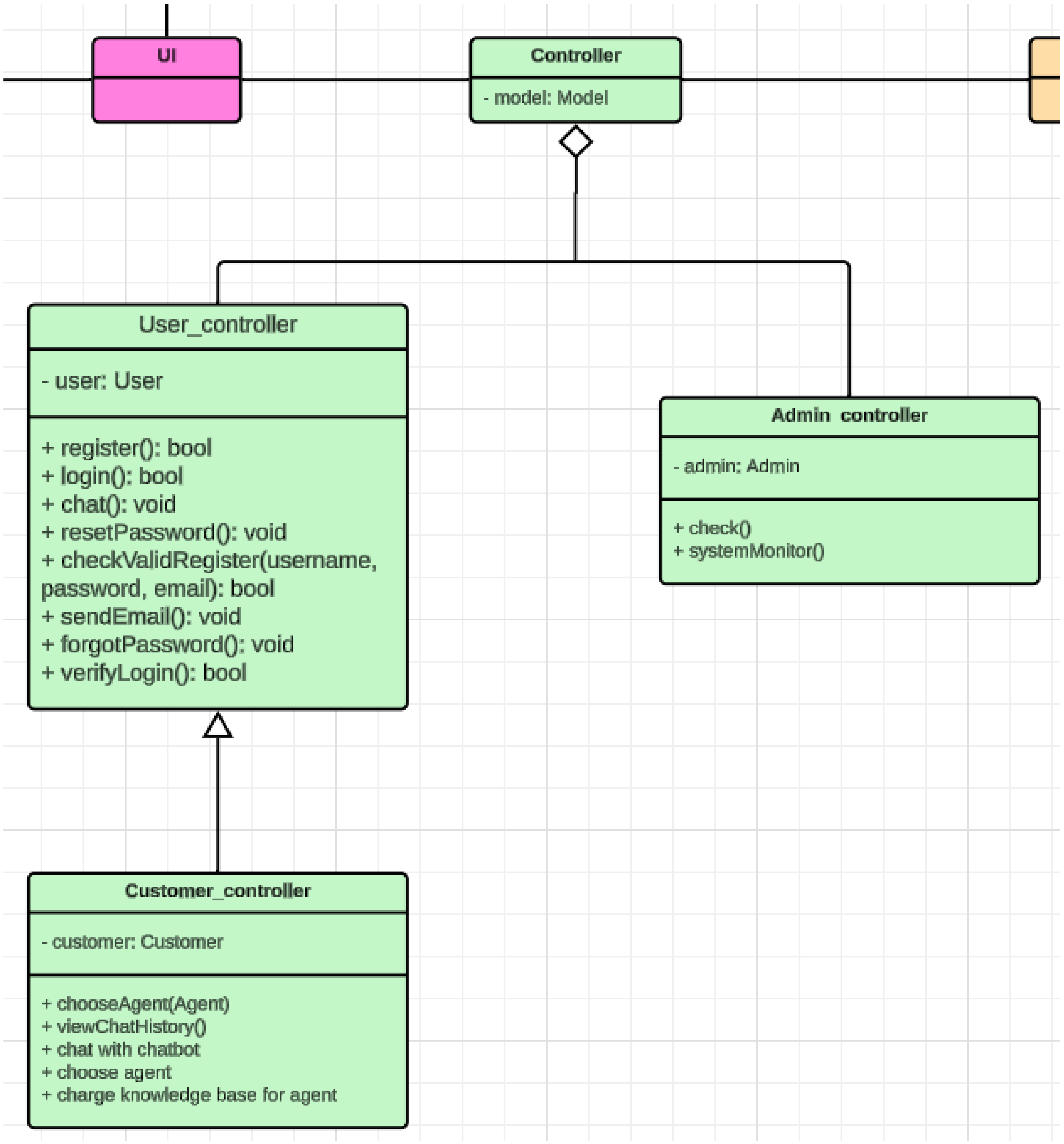
Conversation

- Has an aggregation relationship to **User**. (each User can have from 1 to many conversations).
- Responsible for handling conversations between users and agents.
- Stores a list of messages and provides methods to save/load conversations.

Message

- Has a composition relationship to **Conversation**. (each Conversation can have from 1 to many messages).
- Represents individual messages in a conversation.
- Includes attributes for sender, content, and message ID.

4.2 Component: Controller



User_Controller

- Responsibilities:
  - Manage user-related actions such as registration, login, password reset, and communication.
  - Handles the validation of registration details and login verification.
  - Send emails for actions like registration or password recovery.
- Methods:
  - register(): Registers a new user.
  - login(): Authenticates the user.

- chat(): Initiates a chat with an agent.
- resetPassword(): Resets the user’s password.
- checkValidRegister(): Validates registration details such as username, password, and email.
- sendEmail(): Sends an email (e.g., for registration confirmation).
- forgotPassword(): Initiates the process for resetting the password.
- verifyLogin(): Verifies the login credentials of the user.

Customer Controller

- Responsibilities:
  - Manages customer-specific actions such as choosing an agent, viewing chat history, and interacting with the chatbot.
  - Handles the management of knowledge bases for agents.
- Methods:
  - chooseAgent(Agent): Allows the customer to select an agent.
  - viewChatHistory(): Displays the customer's chat history.
  - chat(): Initiates a chat with the selected agent or chatbot.
  - chooseAgent(): Allows customers to choose which agent they want to interact with.
  - chargeKnowledgeBaseForAgent(): Charges or updates the knowledge base for the selected agent.

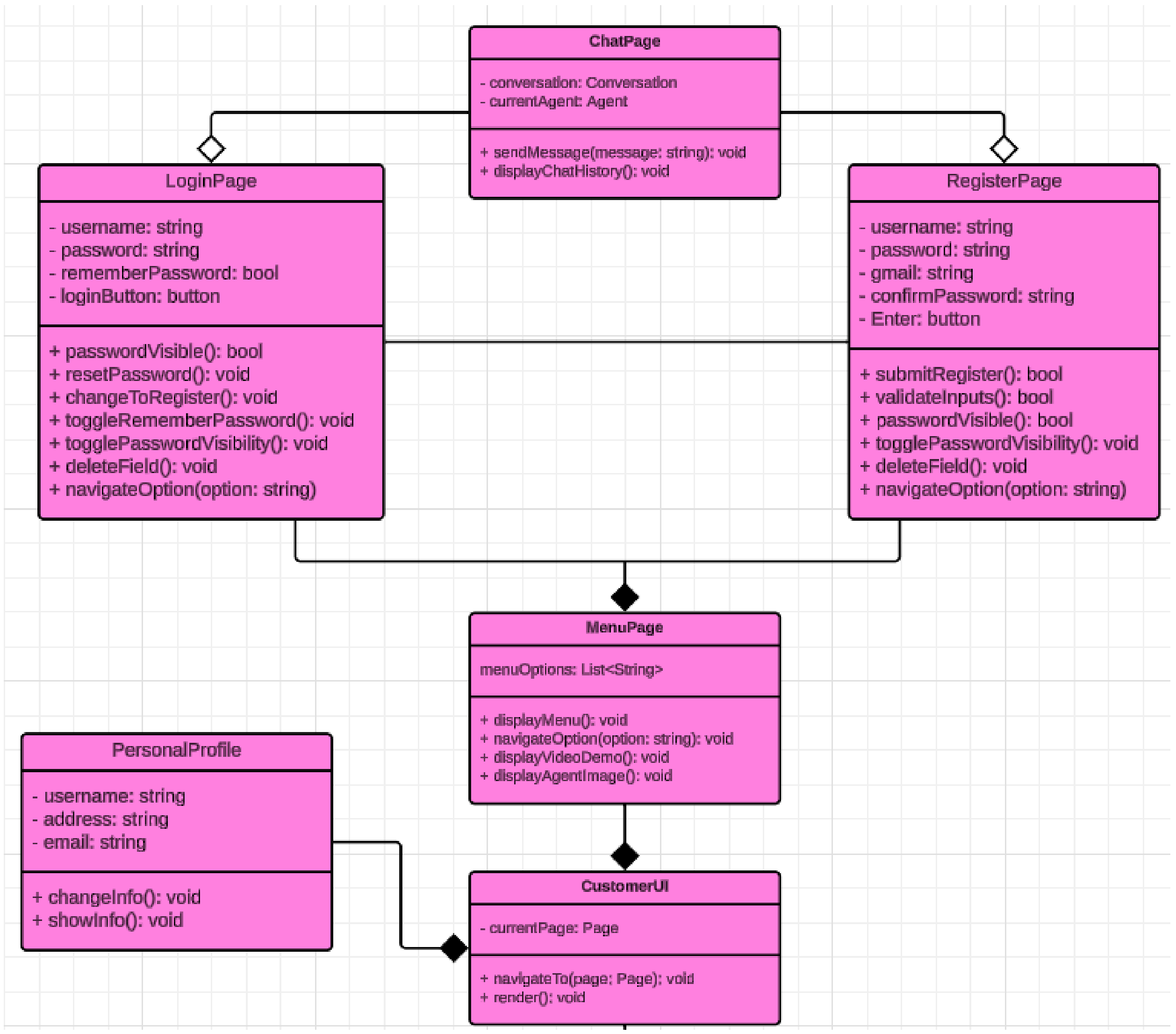
Admin Controller

- Responsibilities:
  - Manages admin-specific actions such as checking and verifying the system.
- Methods:
  - check(): Verifies the system or performs checks on the application.
  - systemMonitor(): Allows the administrator to access the **SystemMonitor** module to view the system status, logs, and other monitoring tools.

4.3 Component: UI

The **UI components** handle the interaction between users and the system, providing distinct interfaces for different user roles (customers, admins). These components ensure a seamless user experience for accessing the application's features and functionality.

4.3.1: CustomerUI


ChatPage
- conversation: Conversation
- currentAgent: Agent
+ sendMessage(message: string): void
+ displayChatHistory(): void
LoginPage
- username: string
- password: string
- rememberPassword: bool
- loginButton: button
+ passwordVisible(): bool
+ resetPassword(): void
+ changeToRegister(): void
+ toggleRememberPassword(): void
+ togglePasswordVisibility(): void
+ deleteField(): void
+ navigateOption(option: string)
RegisterPage
- username: string
- password: string
- gmail: string
- confirmPassword: string
- Enter: button
+ submitRegister(): bool
+ validateInputs(): bool
+ passwordVisible(): bool
+ togglePasswordVisibility(): void
+ deleteField(): void
+ navigateOption(option: string)
MenuPage
menuOptions: List<String>
+ displayMenu(): void
+ navigateOption(option: string): void
+ displayVideoDemo(): void
+ displayAgentImage(): void
PersonalProfile
- username: string
- address: string
- email: string
+ changeInfo(): void
+ showInfo(): void
CustomerUI
- currentPage: Page
+ navigateTo(page: Page): void
+ render(): void

CustomerUI

- Responsibilities:
  - Acts as the main interface for the customer to interact with the application.
  - Handles navigation between different pages (LoginPage, RegisterPage, ChatPage, etc.).
  - Renders the corresponding UI components for each page.
- Methods:
  - navigateTo(page: Page): Navigates to the specified page.
  - render(): Renders the current page.

LoginPage

- Responsibilities:
  - Provides the interface for users to log in to the system.
  - Allows password visibility toggling and remembers login credentials if selected.

- Facilitates navigation to the registration page or password reset process.
- Methods:
  - passwordVisible(): Toggles the visibility of the password field.
  - resetPassword(): Initiates the password reset process.
  - changeToRegister(): Redirects the user to the RegisterPage.
  - toggleRememberPassword(): Enables or disables the "Remember Password" option.
  - togglePasswordVisibility(): Toggles between showing or hiding the password.
  - deleteField(): Clears the content of input fields.
  - navigateOption(option: string): Navigates to other options (e.g., RegisterPage or Forgot Password).

RegisterPage

- Responsibilities:
  - Provides the interface for new users to register an account.
  - Validates user inputs and ensures all required fields are completed correctly.
  - Facilitates navigation to the LoginPage after successful registration.
- Methods:
  - submitRegister(): Processes the user's registration request.
  - validateInputs(): Validates the user's inputs during registration.
  - passwordVisible(): Toggles the visibility of the password field.
  - togglePasswordVisibility(): Switches between showing or hiding the password.
  - deleteField(): Clears input fields.
  - navigateOption(option: string): Navigates to other options (e.g., LoginPage).

ChatPage

- Responsibilities:
  - Provides a chat interface for users to communicate with AI agents.
  - Displays the conversation history between the user and the agent.
- Methods:
  - sendMessage(message: string): Sends a message from the user to the agent.
  - displayChatHistory(): Displays the history of the chat session.

MenuPage

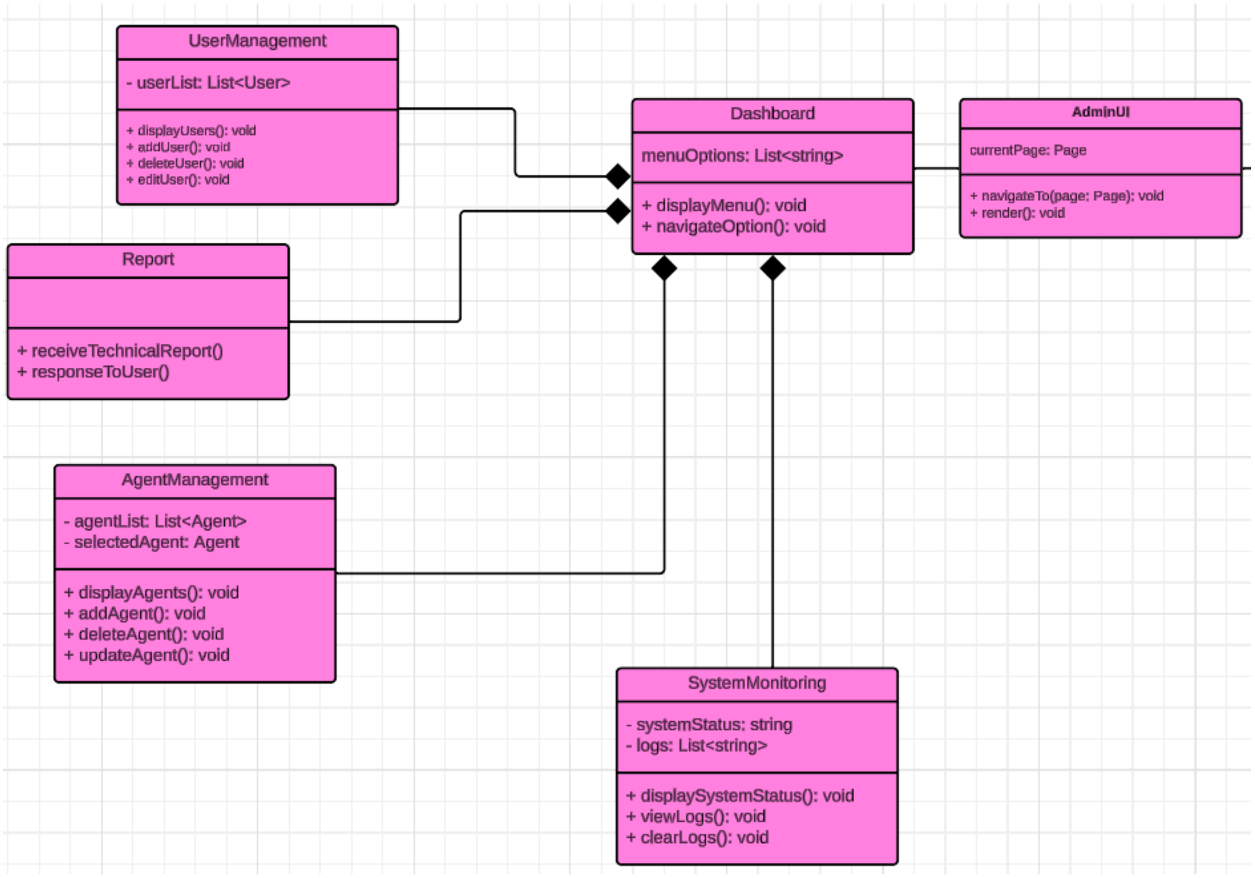
- Responsibilities:
  - Serves as the central navigation menu for the customer, offering options like video demos, agent information, and other functionalities.
- Methods:
  - displayMenu(): Displays the list of menu options.
  - navigateOption(option: string): Redirects the user to the selected menu option.
  - displayVideoDemo(): Plays a demo video for the application or agents.
  - displayAgentImage(): Shows images or information about the AI agent.

PersonalProfile

- Responsibilities:
  - Displays and manages the customer's profile information, such as username, email, and address.
  - Allow the customer to update their information if needed.
- Methods:
  - changeInfo(): Updates the customer's profile information.

- showInfo(): Displays the current profile information.

4.3.2: AdminUI



AdminUI acts as the primary interface for administrators to manage the system and handles navigation across various administrative modules such as User Management, Agent Management, System Monitoring, and reports of user

UserManagement

- Responsibilities:
  - Handles all user-related operations, such as adding, editing, deleting, and viewing users.
  - Maintains a list of registered users for administrative oversight.
- Attributes:
  - userList: List<User>: A list containing all user profiles.
- Methods:
  - displayUsers(): Displays the list of users.
  - addUser(): Adds a new user to the system.
  - deleteUser(): Deletes a user from the system.
  - editUser(): Edits the information of an existing user.

AgentManagement

- Responsibilities:
  - Manages the AI agents, including adding, updating, or deleting them.
  - Enables the administrator to view the list of agents and manage their configurations.
- Attributes:
  - agentList: List<Agent>: A list containing all AI agents.

- selectedAgent: Agent: Represents the currently selected agent for operations.
- Methods:
  - displayAgents(): Displays the list of agents.
  - addAgent(): Adds a new AI agent to the system.
  - deleteAgent(): Removes an existing agent from the system.
  - updateAgent(): Updates the configuration of an agent.

SystemMonitoring

- Responsibilities:
  - Monitors the system's operational status, including logs and system performance.
  - Provides functionality for viewing and clearing logs.
- Attributes:
  - systemStatus: string: Displays the current status of the system.
  - logs: List<string>: A collection of system logs for tracking events and errors.
- Methods:
  - displaySystemStatus(): Shows the current status of the system.
  - viewLogs(): Displays the system logs.
  - clearLogs(): Clears the system logs.

Report

- Responsibilities:
  - Manages the reporting process, including receiving and responding to technical reports submitted by users.
- Methods:
  - receiveTechnicalReport(): Receives technical reports from users or the system.
  - responseToUser(): Sends a response to the user after reviewing a report.

5. Deployment

5.1 Web Browser (Client Node)

- **Purpose**:
  - Acts as the client-side interface for users to interact with the web application.
  - Renders the HTML, CSS, and JavaScript served by the web server.
  - Sends HTTP/HTTPS requests to fetch data and interact with APIs hosted on the backend.
- **Details**:
  - Runs on various platforms such as desktops, laptops, tablets, and smartphones.
  - Common browsers include Google Chrome, Mozilla Firefox, Safari, and Microsoft Edge.
- **Responsibilities**:
  - Displays the user interface (UI).
  - Sends requests to the web server when users navigate the app or interact with features.
  - Receives and renders responses (HTML pages or JSON data for single-page applications).
- **Communication**:
  - Communicates with the web server over **HTTPS** to ensure secure transmission of data.

5.2 Web Server (Front-End Node)

- **Purpose**:
  - Hosts static files (HTML, CSS, JavaScript, images).
  - Acts as the middleman between the web browser and the application server by routing API requests.
- **Details**:

- Uses web server software like **Nginx**, **Apache**, or a CDN service.
- Can be optimized to improve performance with caching, compression, and load balancing.

- **Responsibilities**:
  - Serves static assets to the client quickly and reliably.
  - Routes dynamic API requests to the application server.
  - Enhances scalability by working with CDNs and load balancers.
- **Communication**:
  - **HTTPS**: Receives requests from the web browser and serves static content.
  - **HTTPS**: Forwards dynamic requests (e.g., API calls) to the application server.

5.3 Application Server (Back-End Node)

- **Purpose**:
  - Executes business logic and processes requests from the client.
  - Hosts the **Node.js Express** framework to handle API endpoints, middleware, and application logic.
- **Details**:
  - Runs on a server (e.g., AWS EC2, DigitalOcean, or Heroku).
  - Implements security features like authentication (JWT, OAuth) and authorization.
- **Responsibilities**:
  - Handles API requests (e.g., CRUD operations for database interactions).
  - Validates, processes, and responds to client data.
  - Manages sessions, cookies, or tokens for secure access.
  - Integrates with external services (e.g., third-party APIs, payment gateways).
- **Communication**:
  - **HTTPS**: Receives API calls from the web server.
  - **Secure DB Connection**: Queries and updates the database.

5.4 Database Server

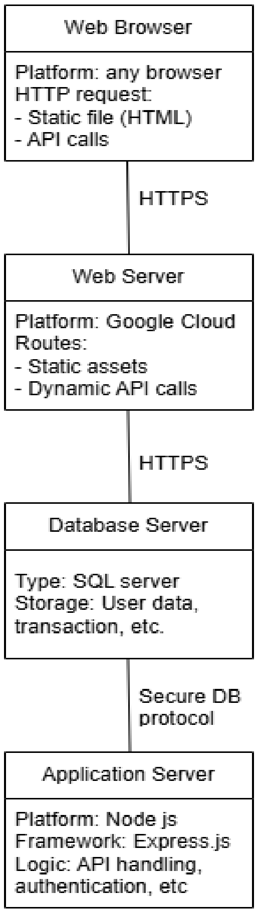
- **Purpose**:
  - Provides persistent storage for the web application’s data.
- **Details**:
  - Common technologies include **MySQL**, **PostgreSQL**, **MongoDB**, or **SQLite**.
  - Runs on a separate physical or virtual server, or uses a managed cloud database service (e.g., AWS RDS, Google Cloud SQL).
  - Data includes user profiles, transactions, logs, configurations, and more.
- **Responsibilities**:
  - Stores, retrieves, and updates application data as requested by the application server.
  - Ensures data integrity and consistency with ACID properties (for relational databases).
  - Supports backups and disaster recovery processes.
  - Enforces access control policies to restrict unauthorized database access.
- **Communication**:

- **Secure Database Protocol (e.g., TLS)**: Communicates securely with the application server.
- Firewalls and IP whitelisting protect the database from unauthorized access.

5.5 Relationships Between Nodes (Details)

1. **Web Browser ↔ Web Server**:
    - **Protocol**: HTTPS.
    - The web browser sends requests for static files (e.g., JavaScript, CSS) and API data.
    - The web server responds with the requested files or routes the API request to the application server.
2. **Web Server ↔ Application Server**:
    - **Protocol**: HTTPS.
    - The web server forwards dynamic requests (e.g., data queries) to the application server.
    - The application server processes these requests and responds with JSON, XML, or other formats.
3. **Application Server ↔ Database Server**:
    - **Protocol**: Secure Database Connection (e.g., TLS).
    - The application server queries the database for storing or retrieving application data.
    - The database server responds with the required data or confirmation of updates.

5.6 Diagram:



6. Implementation View

- project-root/
  - docs/ # Documentation for the project (e.g., README.md, API docs, etc.)
  - node_modules/ # Auto-managed Node.js dependencies (npm-installed packages)
  - src/ # Source code folder
    - backend/ # Backend-related files (Express.js)
      - controllers/ # Handles route logic
        - authController.js
        - userController.js
        - productController.js
      - routes/ # API routes
        - authRoutes.js
        - userRoutes.js
        - productRoutes.js
      - models/ # Data models (e.g., Mongoose/Sequelize schemas)
        - userModel.js
        - productModel.js
      - middlewares/ # Custom middleware
        - authMiddleware.js
        - errorHandler.js
      - config/ # Configuration files
        - dbConfig.js # Database connection settings
        - serverConfig.js # General server configuration
        - dotenv file (.env) # Environment variables
      - app.js # Express app initialization
      - server.js # Main entry point for the backend
    - frontend/ # Frontend-specific files
      - assets/ # Static assets
        - images/ # Image files
          - background-image.png
          - logo.png
        - css/ # Stylesheets
          - styles.css
        - js/ # JavaScript files
          - script.js
      - pages/ # HTML pages
        - login.html
        - dashboard.html
        - reset_password.html
        - verify_email.html
    - utils/ # Shared utilities (optional)
      - logger.js # Logging utility
      - helpers.js # General helper functions
  - tests/ # Unit and integration tests
    - backend/ # Backend test files
    - frontend/ # Frontend test files (e.g., Jest or Mocha files)

- logs/ # Logs generated by the app
  - app.log
  - error.log
- public/ # Public assets (served statically by Express) (optional, if serving frontend from Express server)
- .env # Environment variables (e.g., DB credentials, API keys)
- .gitignore # Git ignore rules
- package.json # NPM metadata and scripts
- package-lock.json # Locked dependency versions
- README.md # Project documentation