# DevJam

# ConnectMe
# Software Architecture Document

**Version 2.0**

# Revision History

| Date | Version | Description | Author |
| --- | --- | --- | --- |
| 28/07/2023 | 1.0 | Overview of the application's components and their logical perspective. | DevJam |
| 18/08/2023 | 2.0 | Deployment and implementation completion | DevJam |
| | | | |
| | | | |

# Table of Contents

# Software Architecture Document

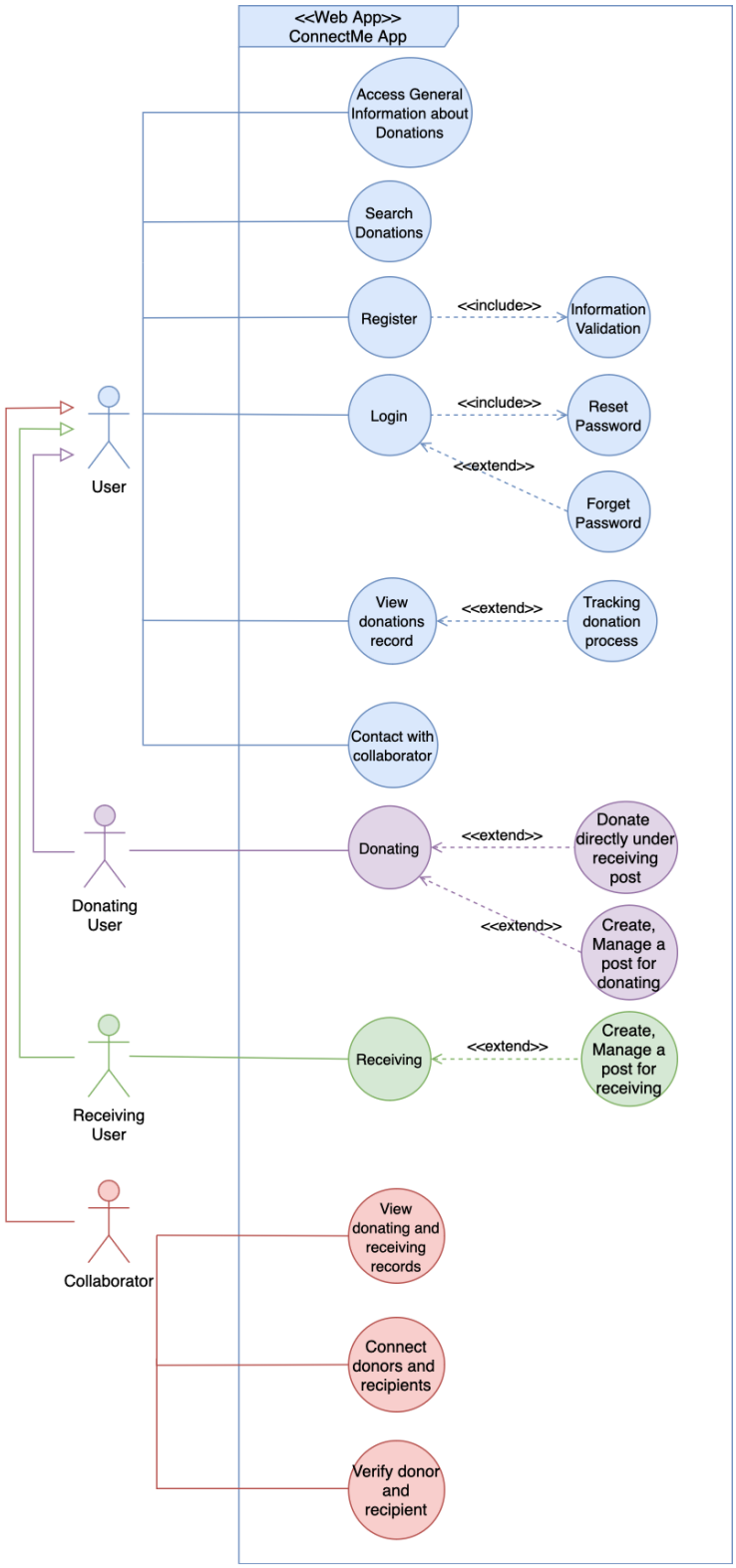
## 1. Introduction

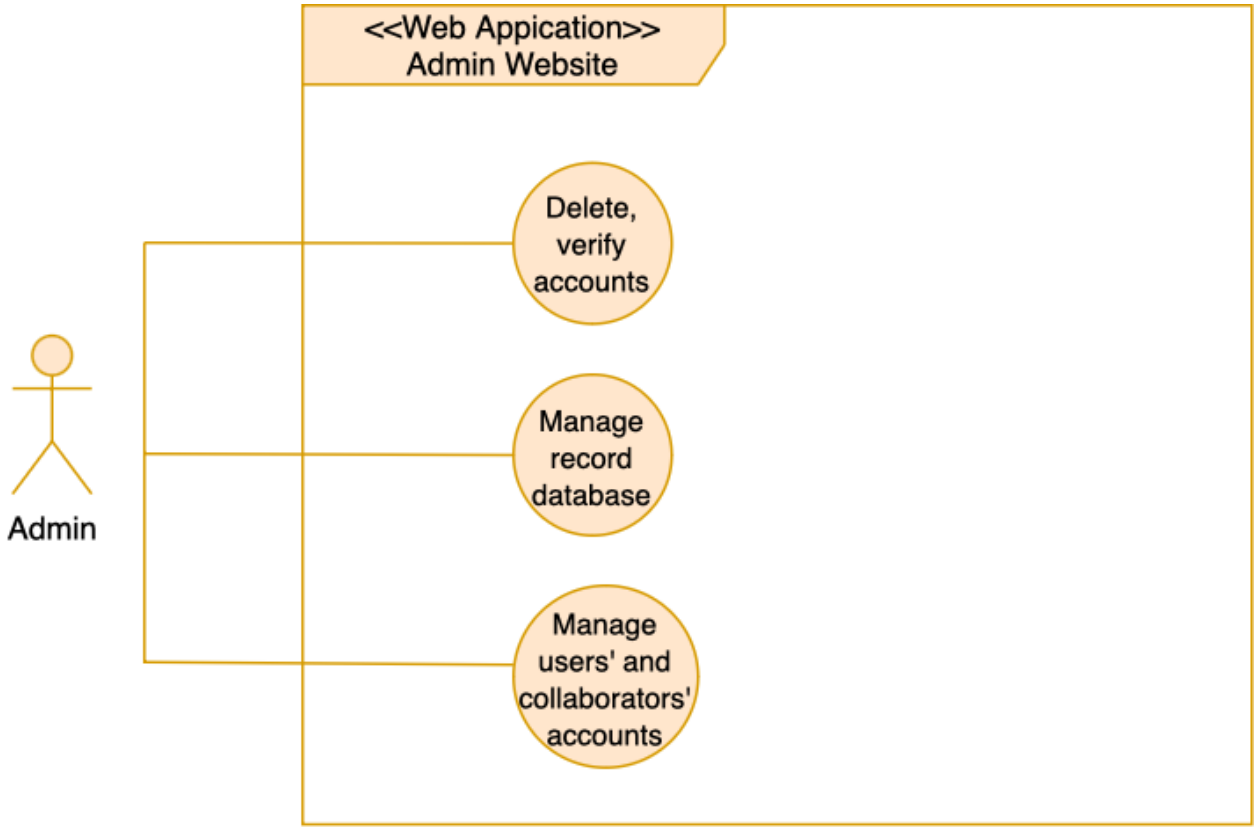
This document will provide an overview of our application's components and their logical perspective, building upon the use-case model outlined in the Use-case specifications document. Additionally, it will demonstrate the application's real-world deployment performance.

## 2. Architectural Goals and Constraints

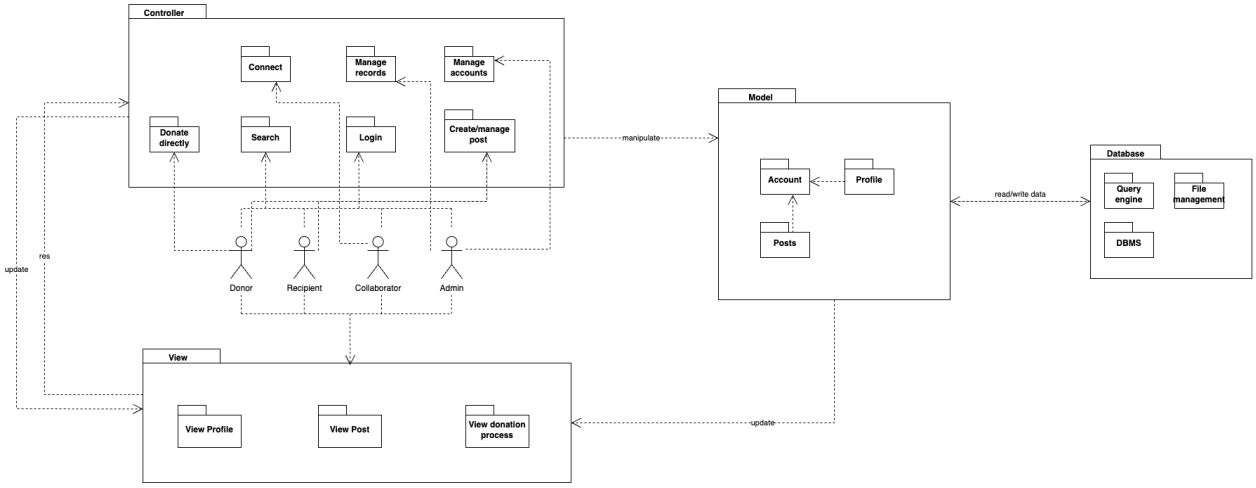
- Technical requirements:
  - Markup and programming language: HTML, CSS, JS, Python.
  - Environment: Web
  - Framework: ReactJS, NodeJS, Django
- Performance requirements:
  - Average response time of 10ms or less for most operations.
  - Quick page load time within 1 second on average.
  - The system should support at least 10,000 concurrent users without significant delays or crashes.
  - User-friendly interface.
- Security:
  - Use Gmail authentication to log in.
  - Forced log out: Users should be automatically logged out after 30 minutes of inactivity to mitigate the risk of data leaks and unauthorized access.
- Reliability:
  - Compatibility with browsers, devices, and platforms.
  - Users can access the application seamlessly from desktops, laptops, tablets, and smartphones.

## 3. Use-Case Model






## 4. Logical View




### a. Client-Server Architecture

- Clients initiate interactions with servers, which offer a range of services. Clients call upon these services whenever necessary and wait for the responses to their requests. The client interface is responsible for displaying data and making minor updates, while the server takes care of data management. The client-server pattern facilitates modifiability and reusability by extracting common services, making it easier to modify them in one place. Additionally, this pattern enhances scalability and availability by centralizing resource and server control.

**b. Model-View-Controller (MVC) Pattern**

- The MVC pattern aims to decouple user interface functionality from application functionality. It achieves this by dividing the application functionality into three distinct components:
    - Models, responsible for handling application data.
    - Views, responsible for displaying data to the user and facilitating user interactions.
    - Controllers, acting as intermediaries between the model and view components, managing state changes.
- By implementing MVC, usability is improved as it enables the separate design and implementation of the user interface from the core application logic.

**i. Client**

- The client acts as a component responsible for requesting services from a server component. Clients are equipped with ports that define the services they need. In the Coop Evaluation System context, the client refers to a web browser utilized to access the system. The client communicates with the server through RESTful web services, utilizing HTTP requests.
- The client's role is to offer users a graphical interface through which they can interact with the system.
- It consists of HTML, CSS, JavaScript, and other associated files that execute within the user's preferred web browser. Additionally, my project incorporates frameworks like Bootstrap for layout and ReactJS. The rationale behind choosing these technologies lies in their utility, large development community, and other advantages they bring to the project.

**ii. Controller**

- The controller encapsulates all business-related logic and manages incoming requests. In many instances, it responds by invoking a method on the model. The view and model are interconnected via a notification mechanism, ensuring that any changes resulting from this action are automatically updated in the view. As a RESTful API server, the server serves as an interface to receive requests from the client and invoke corresponding services.

**iii. Model**

- The component encompasses all the application's business logic, including the logic required for its construction, whether through database queries or other approaches.
- The model incorporates services responsible for preparing data and sending it back to the Controller.

**iv. Database**

- The database is responsible for storing, accessing, and updating data, among many other functionalities. It plays a vital role in managing security and recovery services within the data management system, ensuring the enforcement of constraints in this underlying system.

**v. External API**

- An open or external API is specifically designed to be accessible not only by a broader user base, including web developers, admins, and owners but also by developers within the organization that published the API. Additionally, external developers who wish to use the interface can easily subscribe to it. This openness allows for seamless integration and utilization of the API by both internal and external parties.

**c. Technologies**

**i. Client**

- React with TypeScript (ReactTS) is a declarative, efficient, and versatile JavaScript library for creating user interfaces. It empowers developers to construct intricate UIs by composing small, isolated code units known as "components." When building websites, ReactTS works seamlessly alongside HTML, CSS, and JavaScript, and in our project, we utilize Tailwind CSS for layout.
- Several factors drove the decision to choose React with TypeScript:
    - React with TypeScript is among the most widely adopted JavaScript libraries,

boasting a robust foundation and an active developer community.
    - Leveraging components in ReactTS enables efficient development and enhances code organization.
    - ReactTS significantly contributes to faster app and page load times. It also simplifies maintenance and debugging, ensuring a smoother development experience.
- Vite's lightning-fast development server and native ES module support synergize perfectly with React and TypeScript. Developers enjoy near-instantaneous hot module replacement and faster refresh times, while TypeScript's static typing ensures robust code and early bug detection.

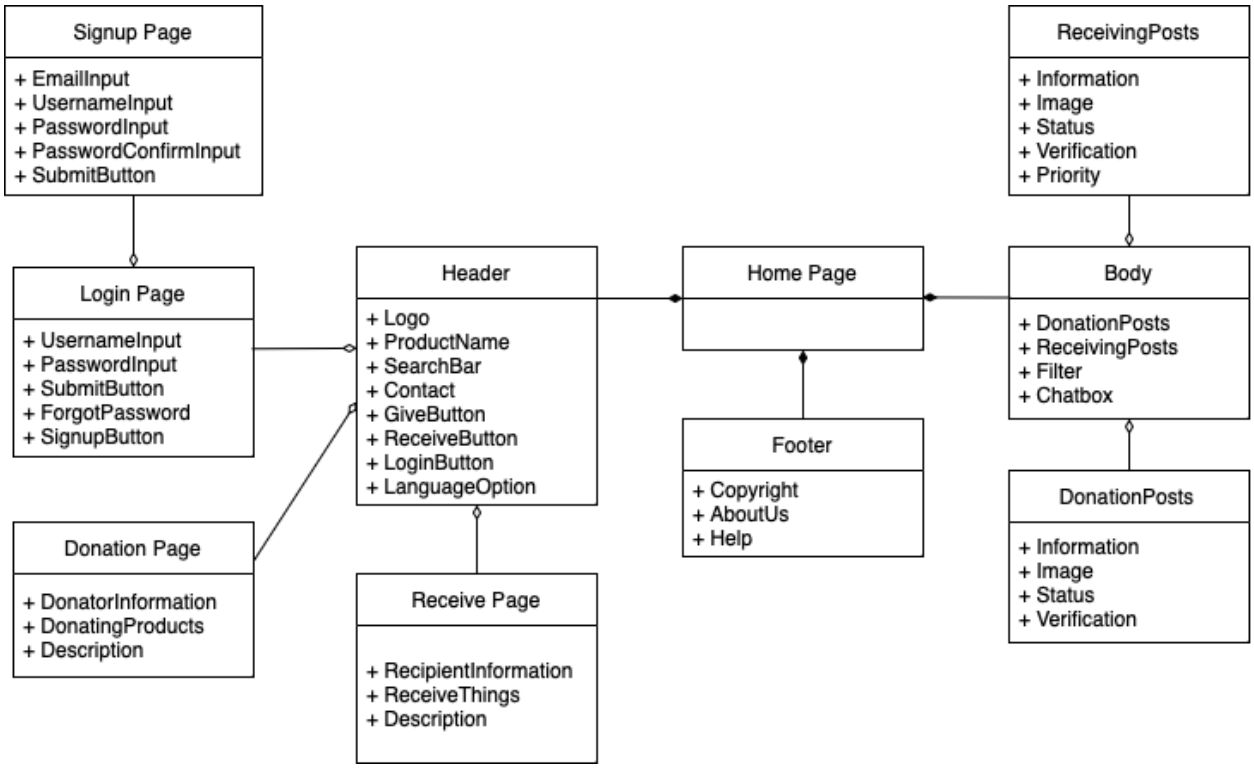
**ii. Server**

- The server acts as a component that accepts and manages requests transmitted by the client, developed within the Node.js environment. Node.js is a cross-platform open-source runtime environment and library that allows web applications to run independently of the client's browser. Furthermore, we utilize the ExpressJS framework, which is a feature-rich Node.js web application framework, offering extensive capabilities for constructing web and mobile applications.
- Reason for choosing ExpressJS:
    - The server functions as a module that receives and handles requests sent by the client, created using the Node.js environment. Node.js is a cross-platform open-source runtime environment and library that enables web applications to operate outside the client's browser.
    - Additionally, we employ the ExpressJS framework, a robust Node.js web application framework that provides a wide range of features for building web and mobile applications.

**iii. Database**

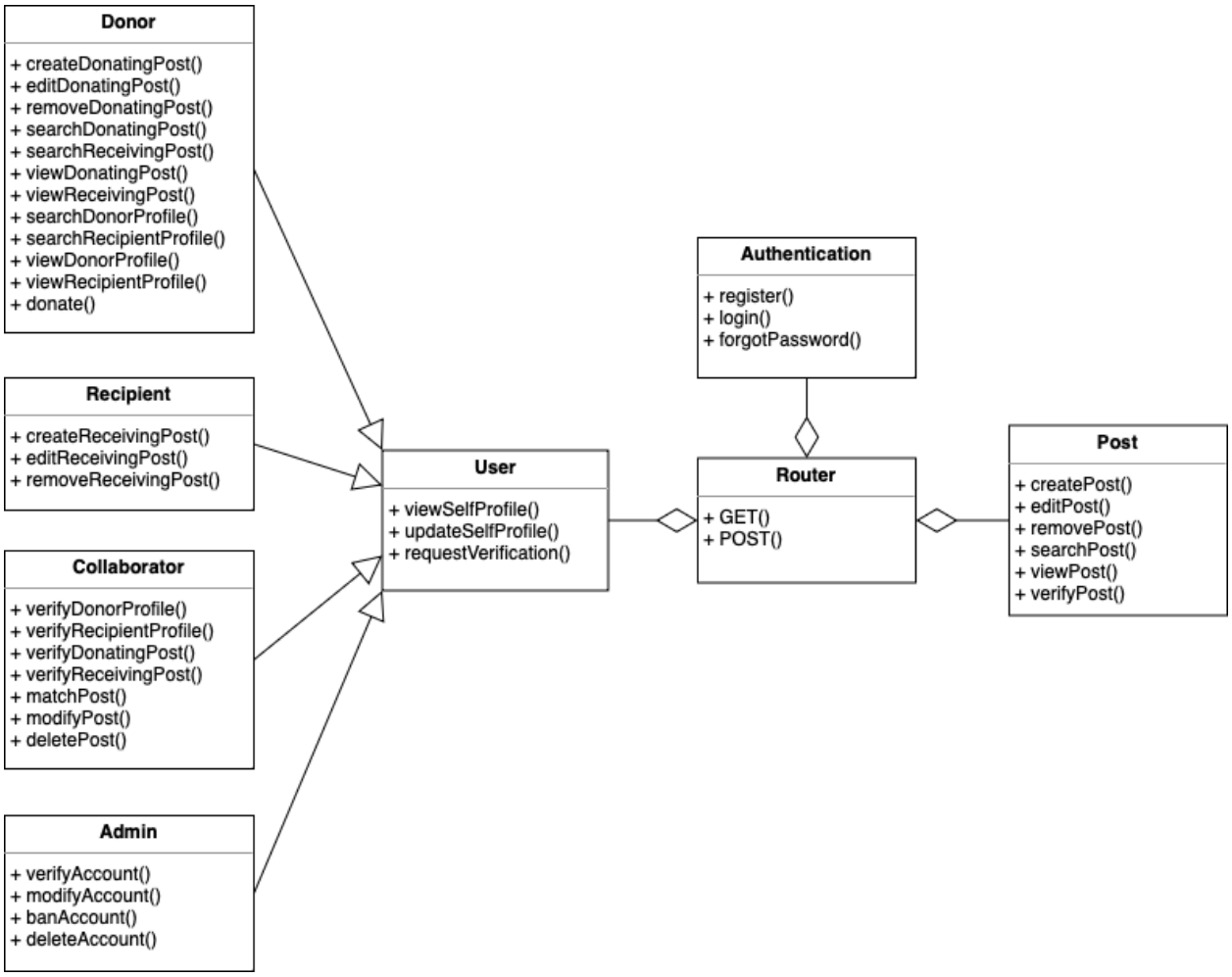
- MongoDB is a popular NoSQL database known for its flexibility and scalability. It stores data in JSON-like documents, making it ideal for handling unstructured or semi-structured data. Some advantages of MongoDB include:
    - Flexible Data Model: MongoDB's document-based model allows developers to store data in flexible and schema-less JSON-like documents. This flexibility enables easy handling of unstructured or semi-structured data, making it well-suited for dynamic and evolving application requirements.
    - Scalability: MongoDB excels in scalability, allowing data distribution across multiple servers through sharding. As data volume grows, MongoDB can seamlessly handle increased traffic and ensure high performance and responsiveness.
    - Rich Query Language: MongoDB provides a powerful and expressive query language, enabling developers to perform complex queries and manipulations on the stored data. This versatility supports a wide range of use cases and enhances data retrieval efficiency.
    - High Performance: MongoDB is designed for efficiency, offering efficient indexing and optimized storage. This results in excellent read-and-write performance, making it ideal for high-throughput applications and real-time data processing.

### 4.1 Component: View




- The View component is used for presenting the data of the model to the user. This component deals with how to link up with the model's data but doesn't provide any logic regarding what this data is all about or how users can use this data. The view's job is to decide what the user will see on their screen, and how.
- The Home Page contains 3 ingredients namely the Header, Footer, and Body. These are typically common sections that appear on multiple pages within the application.
    - The Header includes links to the Login Page, Donating Page, and Receiving Page. These links allow users to navigate to different sections or perform specific actions within the application.
    - The Body contains information about the Donating Posts and Receiving Posts. This section of the Home Page displays content related to donating and receiving activities, such as lists of posts and relevant details.

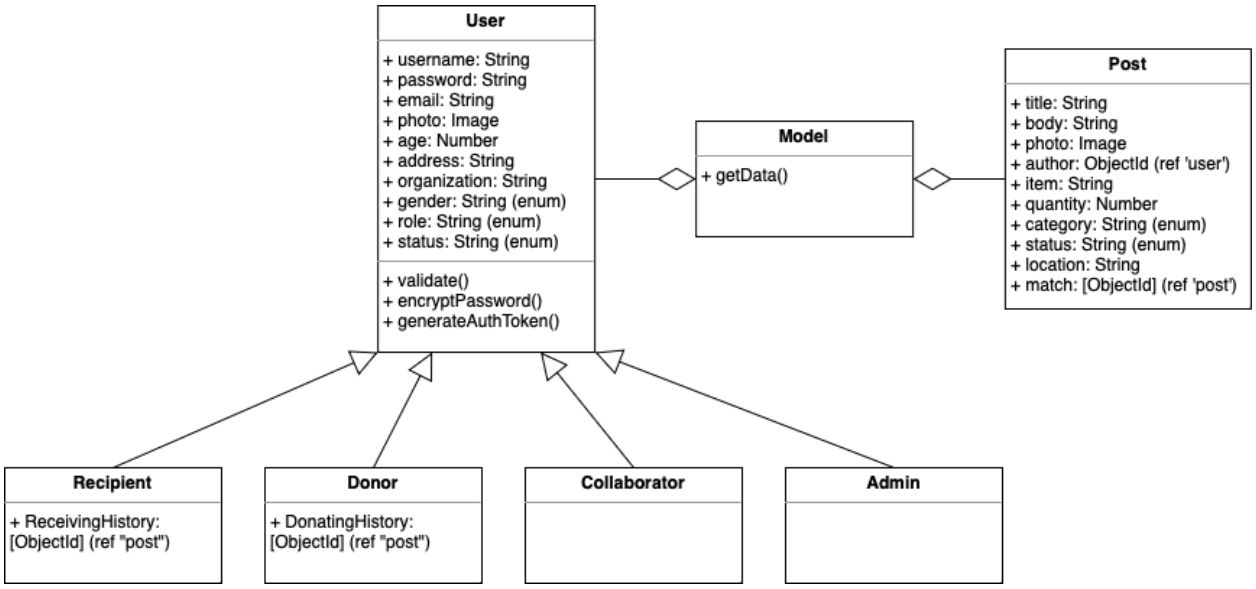
### 4.2 Component: Controller




- The controller's responsibility is to pull, modify, and provide data to the user. Essentially, the controller is the link between the view and model. Through getter and setter functions, the controller pulls data from the model and initializes the views. If there are any updates from the views, it modifies the data with a setter function.
- The "Routes" forward the supported requests (and any information encoded in request URLs) to the appropriate controller functions. The controller functions get the requested data from the models, create an HTML page displaying the data, and return it to the user to view in the browser.
- After receiving a request from the client, the route will navigate to the corresponding controller. After that, call the services in the model and receive data by model and view.
    - **User** Controller: call the general services corresponding to User such as viewing their own profile, updating their own profile, and request for admin's verification.
        - **Donor** Controller: call to the specific services corresponding to Donor such as creating Donating Post, editing Donating Post, removing Donating Post, …
        - **Recipient** Controller: call to the specific services corresponding to Recipient such as creating Receiving Post, editing Receiving Post, removing Receiving Post, …
        - **Collaborator** Controller: call to the specific services corresponding to Collaborator such as verifying Profile, verifying Post, modifying Post, and deleting Post …
        - **Admin** Controller: call to the specific services corresponding to Admin such as creating an account, modifying an account, banning an account, …
    - **Authentication** Controller: call to the general services corresponding to the Authentication process such as Register, Login and Forgot Password.

- **Post** Controller: call to the general services corresponding to Post such as creating a post, editing a post, removing a post, searching for a post, …

### 4.3 Component: Model

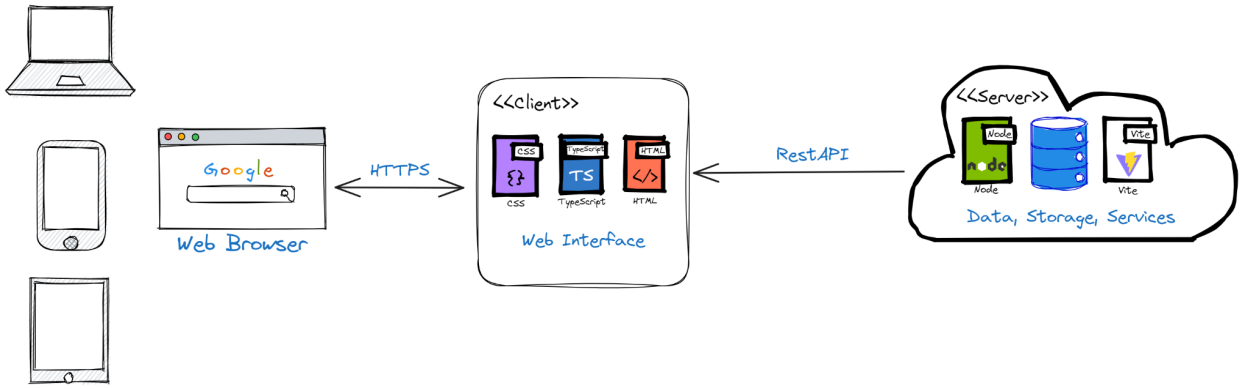



- Model represents the structure of data, the format and the constraints with which it is stored. The model's job is to simply manage the data. Whether the data is from a database, API, or JSON object, the model is responsible for managing it. It maintains the data of the application. Essentially, it is the database part of the application.
    - **User** Model: represents a user entity within the MVC architecture. It encapsulates various attributes and methods related to user data and functionality.
        - username: A string representing the username of the user.
        - password: A string representing the password of the user.
        - email: A string representing the email address of the user.
        - photo: An image object representing the user's photo or profile picture.
        - age: A number representing the age of the user.
        - address: A string representing the address of the user.
        - organization: A string representing the organization or company associated with the user.
        - gender: A string representing the gender of the user. It is defined as an enumeration, which has a limited set of predefined values (male, female, other).
        - role: A string representing the role of the user. Like the gender attribute, it is defined as an enum and has specific values including **admin**, **donor,** and **recipient**.
        - status: A string representing the status of the user. This attribute is also an enum and has values that can be either **verified** or **unverified**.
    - **Recipient** model: is an extension or subclass of the User model, inheriting its attributes and methods. Additionally, the Recipient model introduces a new attribute:
        - ReceivingHistory: An array of ObjectIds referencing the Post model. This attribute represents the receiving history of the recipient, where each ObjectId corresponds to a specific post that the recipient has received.
    - **Donor** model: is also an extension or subclass of the User model, inheriting its attributes and methods. Additionally, the Donor model introduces a new attribute:
        - DonatingHistory: An array of ObjectIds referencing the Post model. This attribute represents the donating history of the donor, where each ObjectId corresponds to a specific post that the donor has made.
    - **Post** Model: represents a post entity within the MVC architecture. It encapsulates various

attributes and methods related to post data and functionality.

- title: A string representing the title or headline of the post.
- body: A string representing the main content or body of the post.
- photo: An image object representing an optional photo associated with the post.
- author: An ObjectId referencing the User model. It represents the author or creator of the post and establishes a relationship between the User and Post models.
- item: A string representing the item or product associated with the post.
- quantity: A number representing the quantity or amount related to the post.
- category: A string representing the category of the post. It is defined as an enum, allowing for a limited set of predefined values (e.g., food, clothing, electronics).
- status: A string representing the status of the post. This attribute is also defined as an enum and can have values such as **published**, **matching,** or **matched**.
- location: A string representing the location associated with the post, such as a physical address or a geographic location.
- match: An array of ObjectIds referencing other Post models. This attribute establishes a relationship between posts, allowing for matching or linking related posts together.

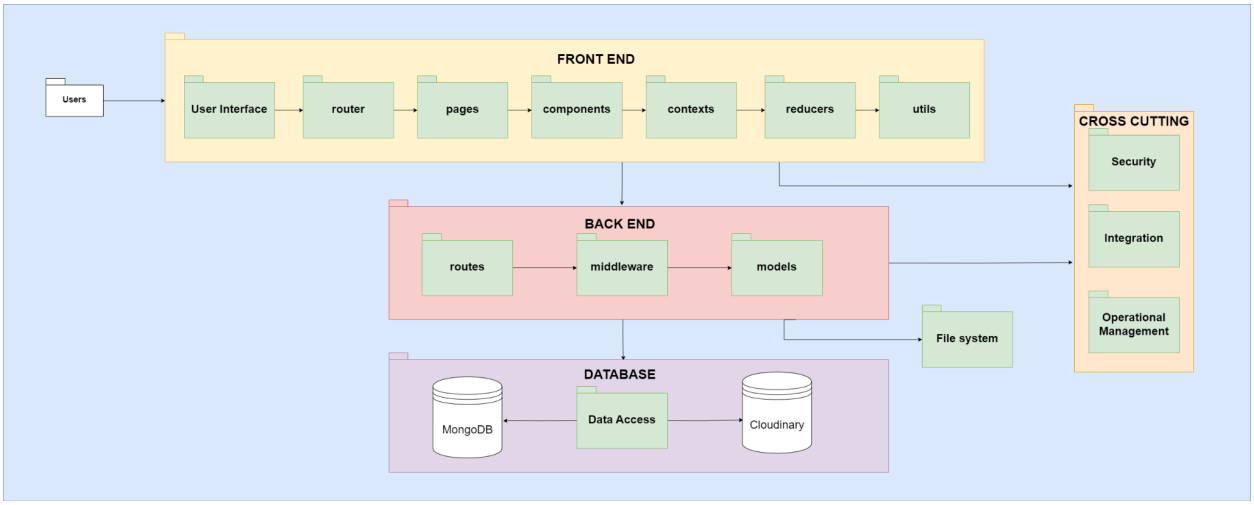
## 5. Deployment




We aim our users can access the web application on any devices (Laptop, PCs, Mobile, Tablet,...), any browsers (Google Chrome, Microsoft Edge, Firefox, Safari,...) and compatible operating systems like Windows, MacOS, Linux.

To cover this, we need to create responsiveness from any scale of screen and use libraries that support mostly browsers. But we will prioritize optimal development on Laptops and PCs first.

# 6. Implementation View




We separated the **Frontend** part into the following subsections:

- The **router folder** can navigate to different pages in the **pages folder**, including Home page, Login page, Donating page, and so on…
- The **page folder** contains several files, each of which defines the layout and components for a specific page.
- The **page folder** can find necessary components in the **components**, **contexts**, **reducers** and **utils folders**.
- The **components folder** includes several subfolders, each of which renders a specific component, such as the Header, Footer, Navbar, Searchbar…
- The **contexts folder** includes files defining the necessary interfaces, and functions, which in turn help to define the user context and post context.
- The **reducer folder** includes files that define the reducer functions, which can be used by the corresponding contexts from the **contexts folders**.

We separated the **Backend** folder into three subfolders:

- The **routes folder** includes the necessary routers to handle the corresponding requests from users.
- The **routes folder** can have access to the **middleware folder**, which defines the necessary middleware functions to help verify users' information or the id of objects.
- The **routes folder** can modify the data stored in the database via the models defined in the **models folder**. There are two main models, which are the user model and the post model, each defining the necessary information to store in each model.

We store data in two separate **database** systems:

- The **MongoDB** database stores information about users and posts.
- The **Cloudinary** database stores images from user profiles and posts updated by users.

The **cross cutting folder** contains three subfolders:

- The **Security Folder** includes information on how we encrypt user passwords using SHA-256 hash algorithms.
- The **Integration Folder** offers information on how we use GitHub and GitLab to integrate our application.
- The **Operational Management** section covers our policies for implementing or testing our application. The File System Folder is where we split and manage our files.