# DitShev

# GoaTalks

# Software Architecture Document

**Version <2.0>**

# Revision History

| Date | Version | Description | Author |
| --- | --- | --- | --- |
| 02/12/2023 | 1.0 | Overview of the application's components and their logical perspective. | DitShev |
| 09/12/2023 | 2.0 | Deployment and implementation completion. | DitShev |
| | | | |
| | | | |

# Table of Contents

# Software Architecture Document

## 1. Introduction

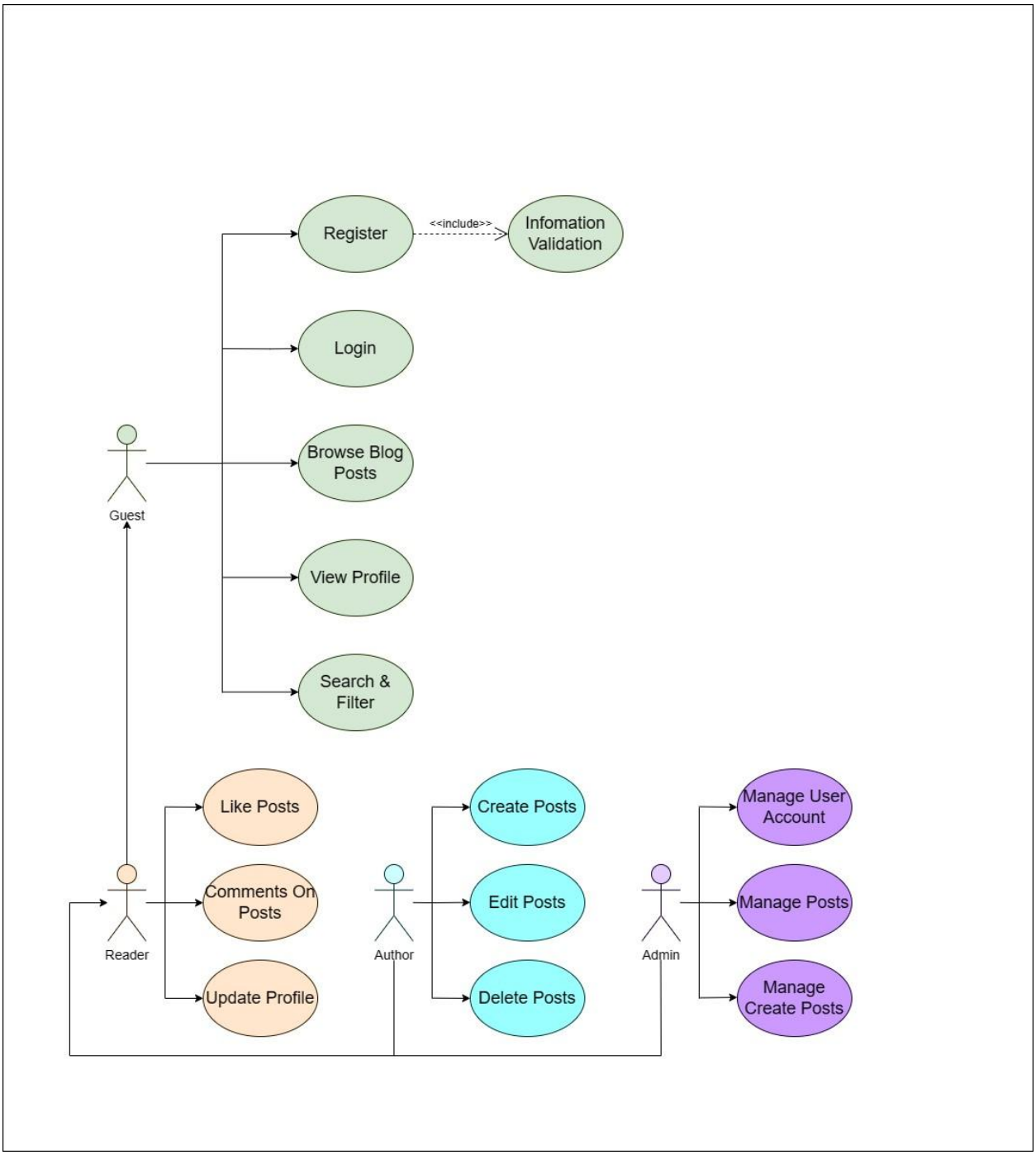
This document serves as a comprehensive guide for understaning the foundational desgin of our blogging platform. Its primary purpose is to provide a reference for developers, project managers, and stakeholders involved in creating, maintaining, and evolving the platform. The document covers the high-level architecture, component interactions, data management, security considerations, and performance optimizations within our blogging ecosystem.

## 2. Architectural Goals and Constraints

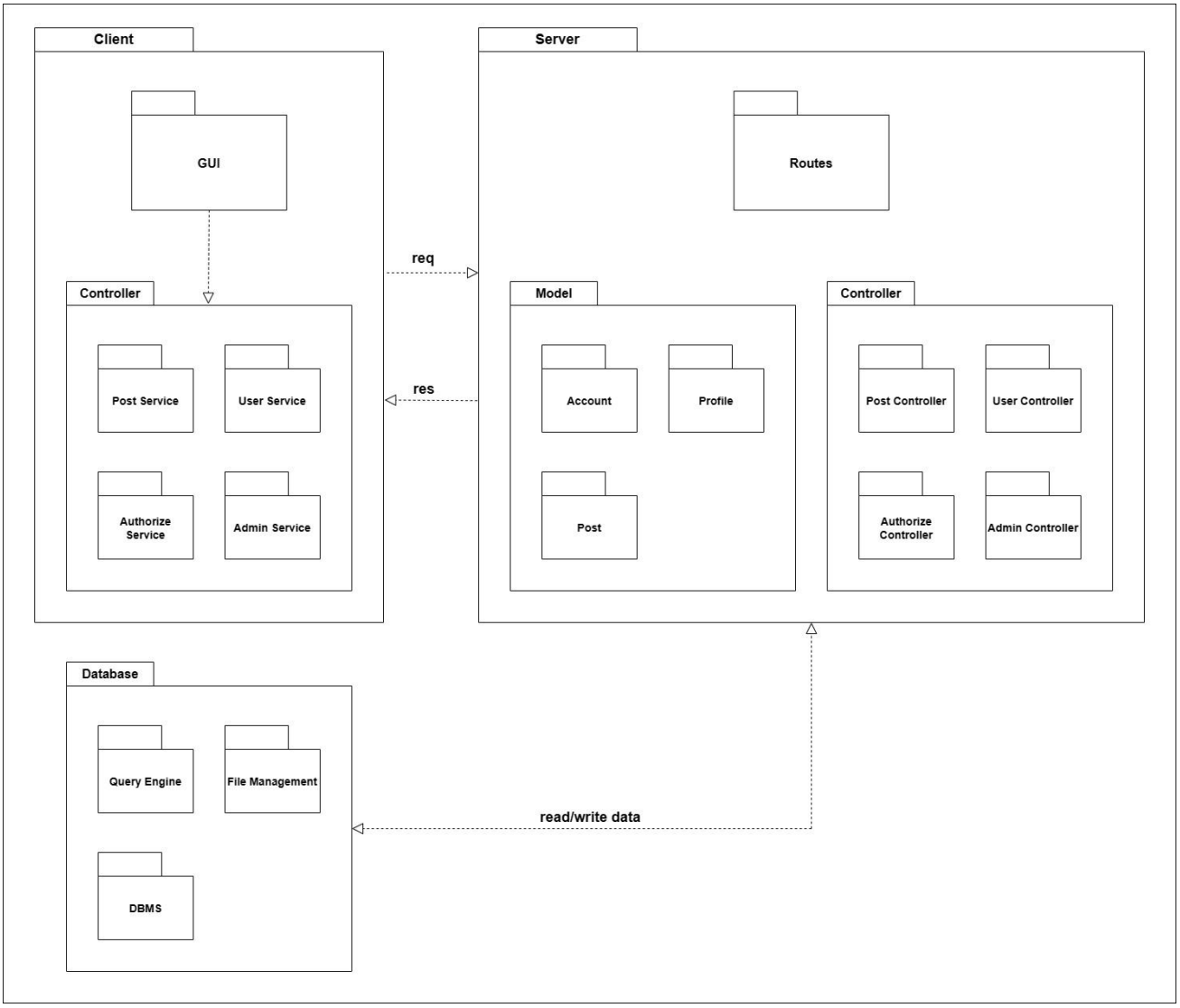
- Technical requirements:
  - Markup and programming languages: HTML, CSS, JS, Python.
  - Environment: Web.
  - Framwork: ReactJS, NodeJS.
- Performance requirements:
  - Implement effective error handling and recovery mechanisms to gracefully manage unexpected issues without causing significant performance degradation.
  - Database queries should execute efficiently, with response times not exceeding a specified threshold.
  - The system should support at least 10,000 concurrent users without significant delays or crashes.
  - User-friendly interface.
- Security:
  - Implement strong password policies, multi-factor authentication, and secure session management for user logins.
  - Employ role-based access controls to ensure that users have appropriate permissions based on their roles.
- Reliability:

- Compatible with various browsers, devices and platforms.
- Prevent overloads on specific servers, optimizing performance and maintaining reliability.
- Implement a strategy for rolling updates to minimize service disruption during maintenance.
- Maintain a positive user experience by handling errors seamlessly and providing informative messages.

# 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

**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**

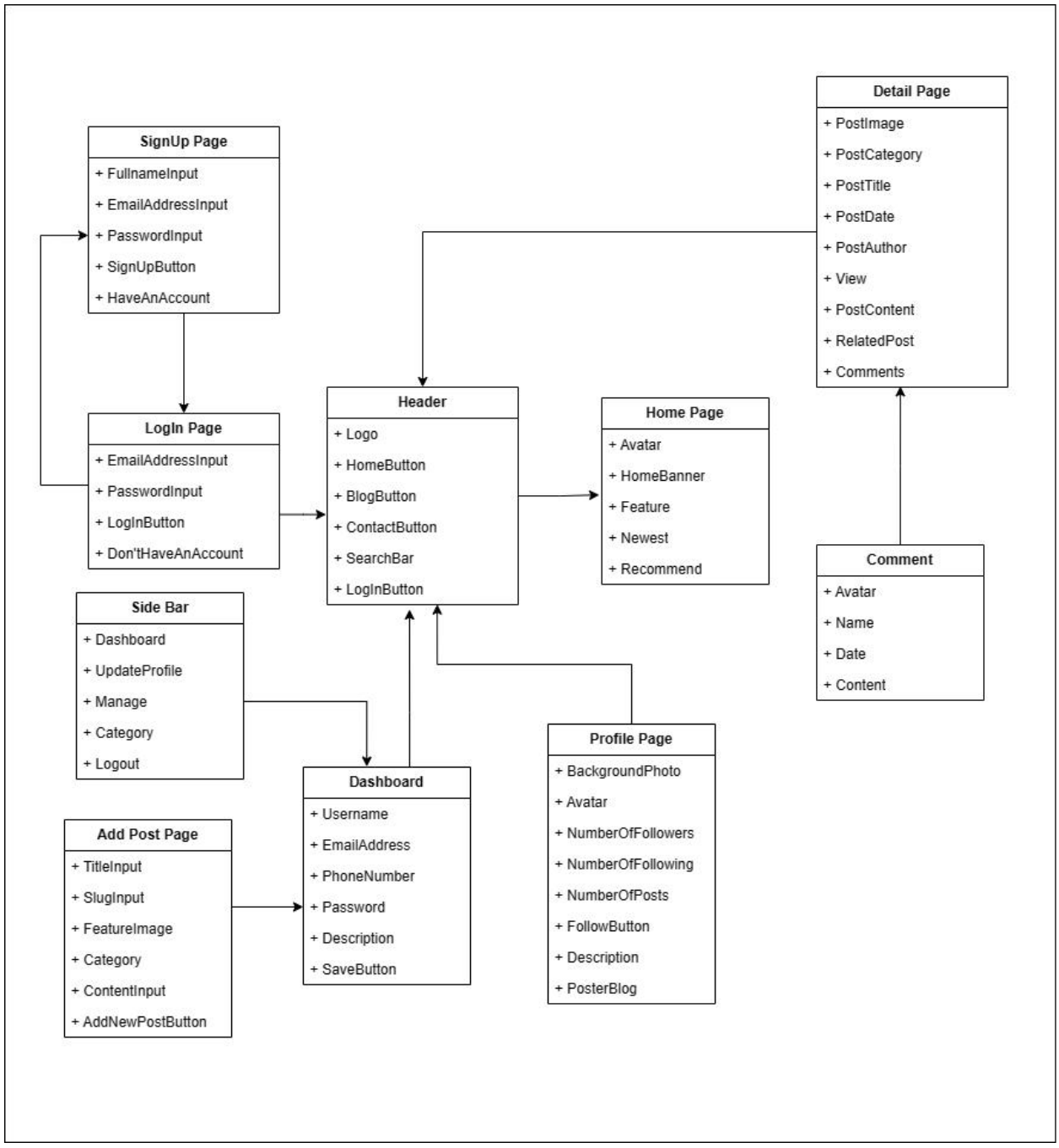
- The server acts as a component that accepts and manages requests transmitted by the client, developed within the Node.js environment. Node.js is a cross-platform open-source runtime environment and library that allows web applications to run independently of the client's browser. Furthermore, we utilize 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: User Interface (ReactJS)

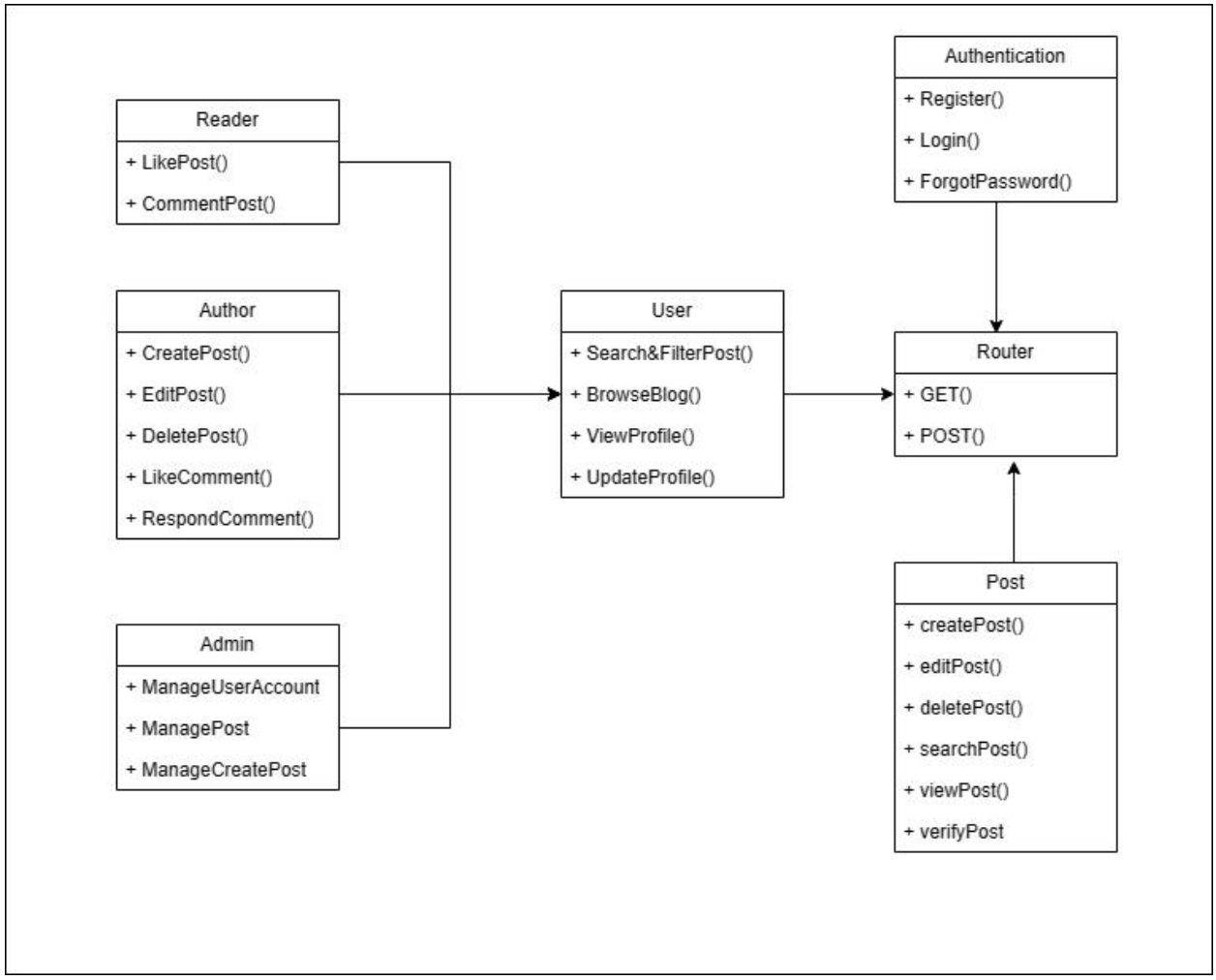



- The View component is used for presenting the data of the model to the user. This component deals with how to link up with the model's data but doesn't provide any logic regarding what this data is all about or how users can use this data. The view's job is to decide what the user will see on their screen, and how.

- The Home Page contains 2 ingredients namely the Header and Body. These are typically common sections that appear on multiple pages within the application.
    - The Header includes links to the Login Page, Blog Page and Contact Page. These links allow users to navigate to different sections or perform specific actions within the application.
    - The Body contains information about Banner, Feature Posts and Newest Posts. This section of the Home Page displays content related to posts, such as lists of posts and relevant details.
- The User Interface (UI) component encompasses the front-end elements of the application built using ReactJS. It primarily focuses on providing a seamless and intuitive interface for users to interact with the system. The responsibilities and key classes within this component include:
    - **Responsibilities:**
        - Rendering various user-facing elements including forms, buttons, navigation bars, etc.
        - Managing state and handling user input effectively.
        - Facilitating communication with the backend services via RESTful APIs or GraphQL.
    - **Key Classes:**
        - App.js: Entry point for the React application, responsible for managing high-level components and routing.
        - Components (e.g., Login, Dashboard, Profile): Represent individual UI elements or sections, each responsible for rendering specific views and handling user interactions.
        - API Services: Wrapper classes or modules for making HTTP requests to backend APIs.
    - **Relationships:**

- Communicates with the Backend Services component via HTTP or HTTPS protocols to fetch or submit data.
- Provides a user-friendly interface to display information retrieved from the server.
- Utilizes CSS for styling and creating an aesthetically pleasing layout.

## 4.2 Component: Backend Services (NodeJS)



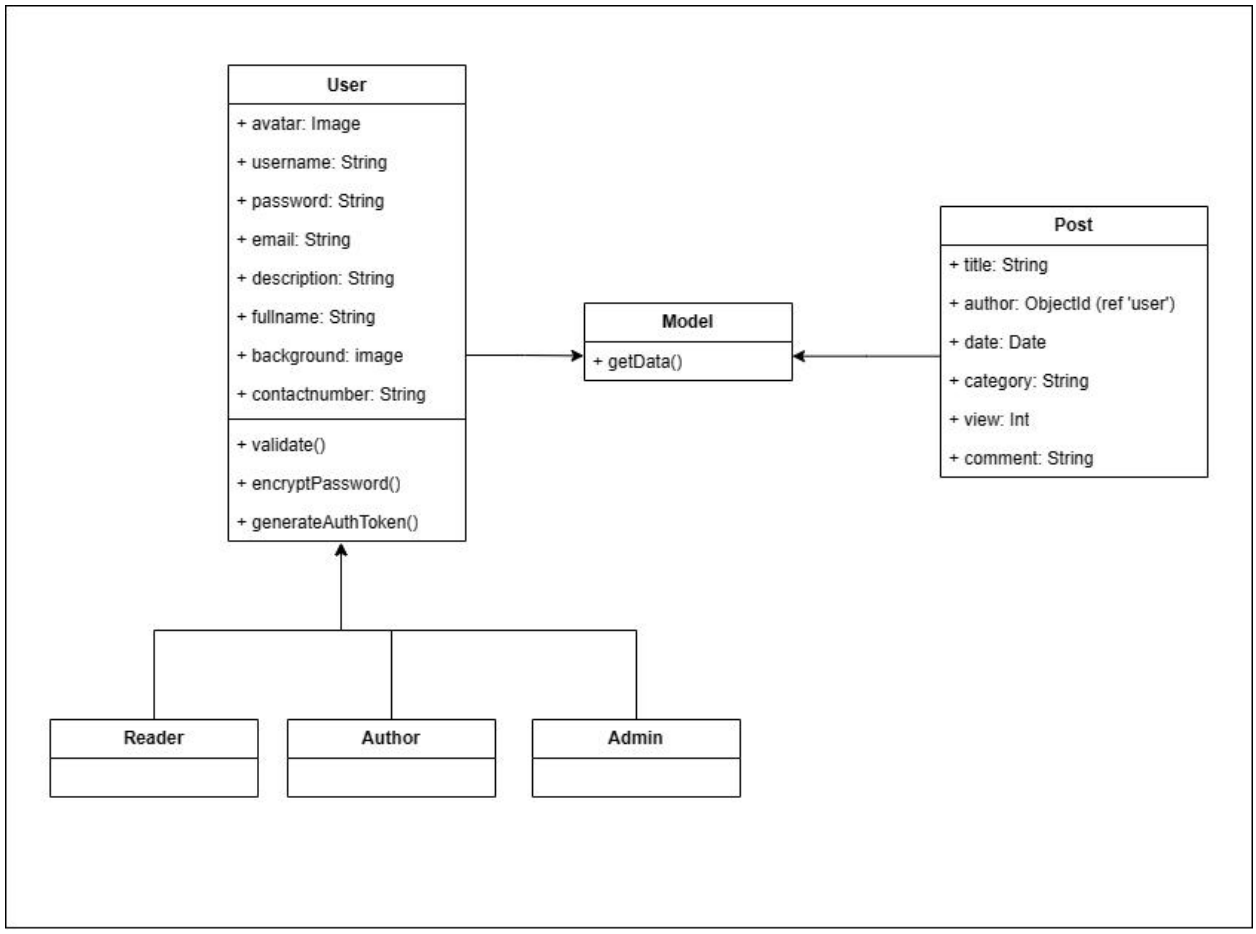

- The Backend Services's responsibility is to provide data to the user. Essentially, the Backend Services is the link between the view and model. Through getter and setter functions, the controller pulls data from the model and initializes the views. If there are any updates from the UI, it modifies 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 Backend Services 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 to the general services corresponding to User such as viewing their own profile, updating their own profile, browsing blog, searching and filtering.
        - **Reader** Controller: call to the specific services corresponding to Reader such as liking posts, comment posts.
        - **Author** Controller: call to the specific services corresponding to Author such as creating post, editing post, deleting post, interating reader’s comments (like and reply).
        - **Admin** Controller: call to the specific services corresponding to Admin such as mananging user account, managing posts, managing create posts.
    - **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, …
- The Backend Services component, developed using NodeJS, constitutes the core logic and functionalities of the system. It interacts with databases, handles business logic, and manages data flow. Its responsibilities and key classes may include:
    - **Responsibilities:**
        - Processing incoming requests from the UI or external services.
        - Implementing the application's business logic, data validation, and manipulation.

- Interacting with the Database component for data retrieval and storage.

- **Key Classes:**
  - Server.js: Main entry point handling routing and middleware.
  - Controllers/Handlers: Modules responsible for processing specific endpoints or functionalities.
  - Data Access Objects (DAOs): Classes or modules for interacting with the database.
- **Relationships:**
  - Communicates with the UI component through RESTful APIs, utilizing HTTP or HTTPS protocols.
  - Interacts with the Database component to perform CRUD operations and retrieve/store data.
  - Enforces security measures like authentication and authorization before processing requests.

### 4.3 Component: Database (MongoDB)




- Database 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 a 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.
    - avatar: An image object representing the user's photo or profile picture.
    - email: A string representing the email of the user.

- description: A string representing the description of the user.
- fullname: A string representing the full name of the user.
- background: An image representing the user's background.
- contactnumber: A string representing the phone number of the user.

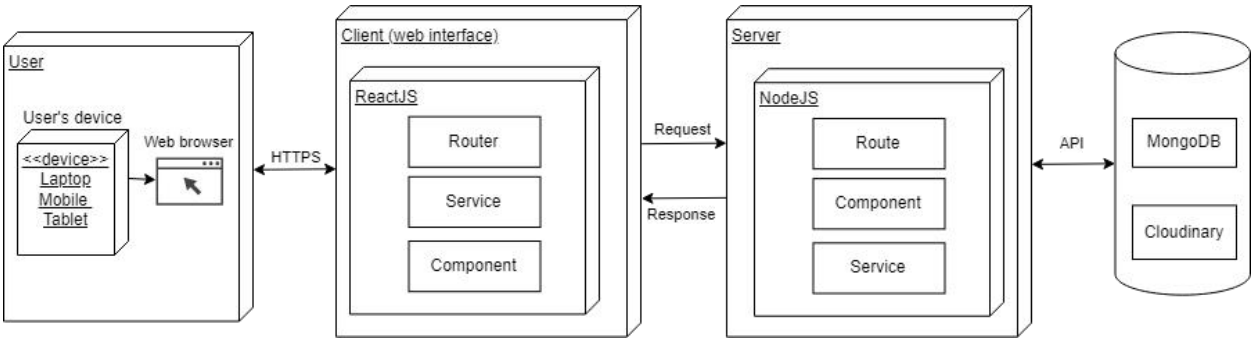
- **Post** Model: represents a post entity within the MVC architecture. It encapsulates various attributes and methods related to post data and functionality.
  - **title**: A string representing the tile of the post.
  - **author**: A string representing the author' id.
  - **date**: A date representing the release date of the post.
  - **category**: A string representing the category of the post.
  - **view**: A number representing the number of view of the post.
  - **comment**: A string representing the comment of the post.

- The Database component in this architecture relies on MongoDB, a NoSQL database, for storing and managing the system's data. MongoDB, being a document-oriented database, operates with collections of JSON-like documents. The details for this component are:
  - **Responsibilities:**
    - Storing data in a flexible, schema-less manner using collections and documents.
    - Handling CRUD (Create, Read, Update, Delete) operations efficiently.
    - **Enabling scalability and high availability through features like sharding and replication.**
  - **Key Classes:**
    - Collections: Represent logical groupings of documents within MongoDB (e.g., Users, Posts).
    - Document Structure: Defines the schema for each document containing fields and values.

- MongoDB Queries: Utilizes MongoDB Query Language (MQL) for executing queries.

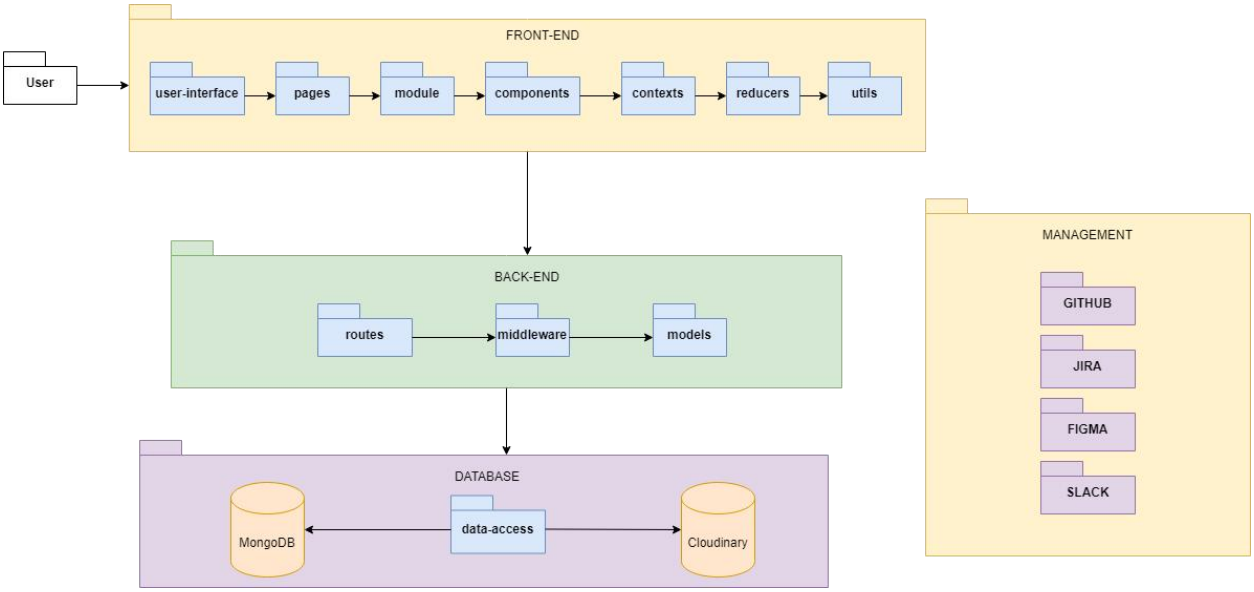
- **Relationships:**
  - Interfaces with the Backend Services component to perform data operations requested by the application.
  - Communicates using the MongoDB query protocol over the network, interacting via the MongoDB driver available for NodeJS.

# 5. Deployment




- User uses personal device to open a web browser (Chrome, Edge,...) and then access to website by calling HTTPs to client (front-end) of the web.
- Client sends requests to server when user needs to get data. Server will call API to the database server to get wanted data. Data will be saved in the database MongoDB, and the images or files will be saved on the Cloudinary service. After receiving data, Server sends responses back to Client, including messages (successful or unsuccessful) and the data. Client gets the responses, handles and displays it to the screen for user.

# 6. Implementation View




We separated the **FRONT-END** part into the following subsections:

- The **pages folder** contains several files, each of which is the design layout and components for a
  specific page (e.g: SignIn, SignUp,Dashboard…).
- The **pages folder** can find necessary components in the **components**, **contexts**, **reducers** and **utils folders**.
- The **module folder** includes several subfolders, each of which is the main functions and components for specific page in **pages folder** (e,g we have folder **post** which is used to define, manage and display posts that will be showed in DetailPage (a page in pages folder)
- The **components folder** is the folder for all settings that will be used (e.g: Button, Checkbox, Dropdown, Form…)
- The **contexts folder** includes files defining the necessary interfaces, and functions, which in turn help to
  define the User context and Post context.
- The **reducers folder** includes files that define the reducer functions, which can be used by the corresponding contexts from the **contexts folders**.
- The **utils folder** is a folder for storing constant components and accessToken of user.

We separated the **BACK-END** folder into three subfolders:

- The **routes folder** includes the necessary routers to handle corresponding requests from users.
- The **routes folder** can have access to the **middleware folder**, which defines the necessary middleware functions to check if user is verified or have specific right.
- The **routes folder** can modify the data stored in the database via the models defined in

the **models**
**folder**. There are two main models: user model and post model, each defines 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.

We also have **MANAGEMENT**, which are services for managing project and discussing with members:

- Github: Is used to manage source code and documents.
- Jira: Service for managing all sprints and personal work of members, manage which work have been done and to be done.
- Slack: Community is used for discussing work and contact with members.
- Figma: A design tool that helps design screens of products.