Manga Geeks

Free Manga Site
Software Architecture Document

Version 1.1

Revision History

Date	Version	Description	Author
18/11/2024	1.0	Fill out section 1.	Hưng
19/11/2024	1.0	Fill out section 3.	Nhân
27/11/2024	1.0	Added model component.	Lạc
28/11/2024	1.0	Added logical view in section 4.	Nhân
30/11/2024	1.0	Added view component.	Bảo
30/11/2024	1.0	Added controller component.	Hưng
12/12/2024	1.1	Redrew view component diagram. Added deployment view in section 5.	Nhân
13/12/2024	1.1	Added implementation view in section 6.	Nhân
14/12/2024	1.1	Adjusted implementation view content.	Hưng

Table of Contents

Software Architecture Document

1. Introduction

This document documents the system's architecture, including design choices, components, diagrams, functional goals, non-functional goals, environments, constraints and specifications.

This document describes the Free Manga Site, colloquially entitled Openbook, which is a website accessible to mobiles and desktops alike to search, read comic books and engage discussion within comment sections. The idea is based on various other similar books-related services, Kindle Store, Wattpad or Yahoo Books to name a few.

From the highest level view, this project uses the MERN tech stack, including:

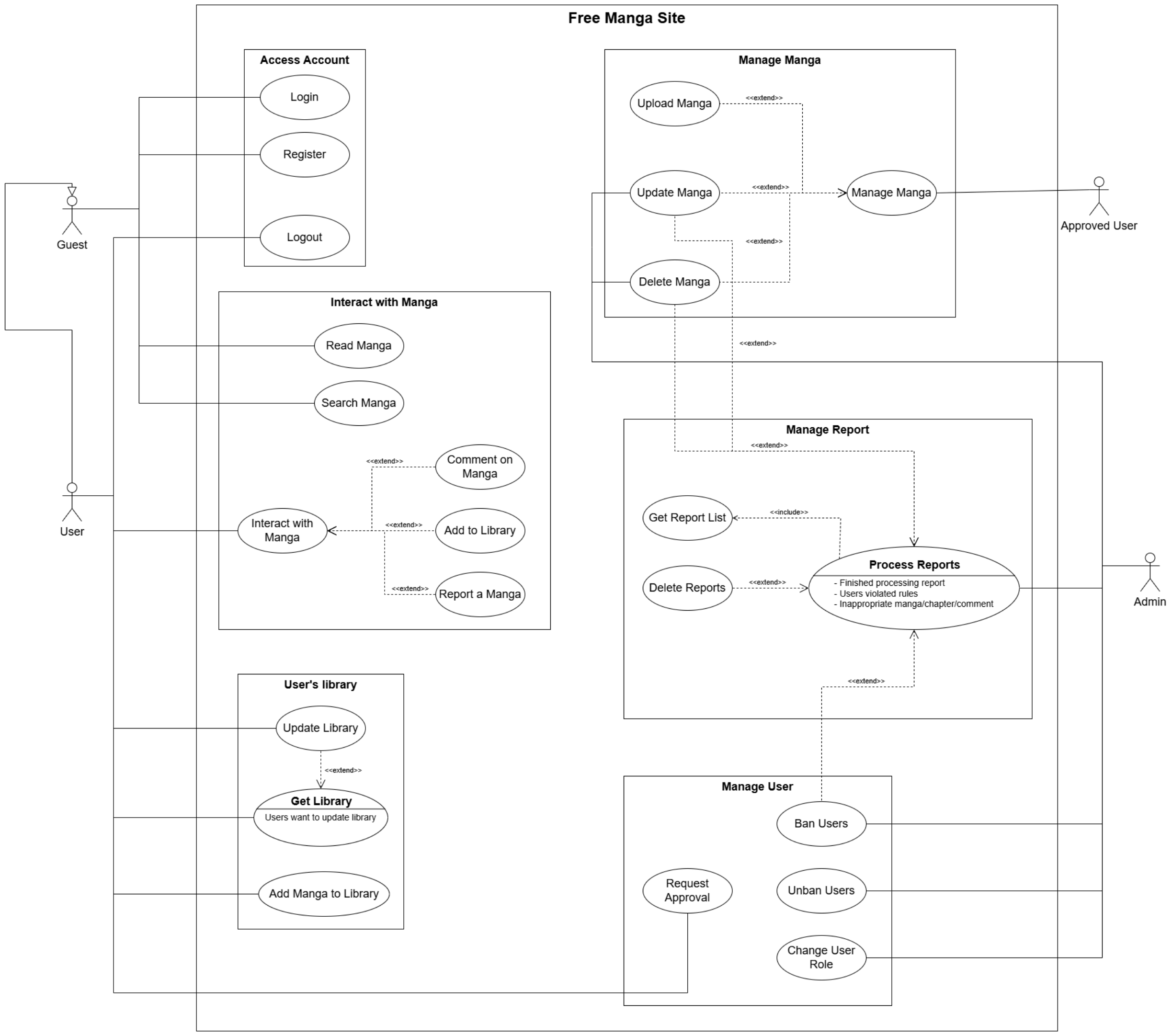
- MongoDB: the document database architecture.
- ExpressJS: for backend API requests handling.
- React: for frontend UI structuring.
- NodeJS: the JavaScript server-side runtime environment for running Express.

For images hosting, we use the cloud platform Cloudinary.

2. Architectural Goals and Constraints

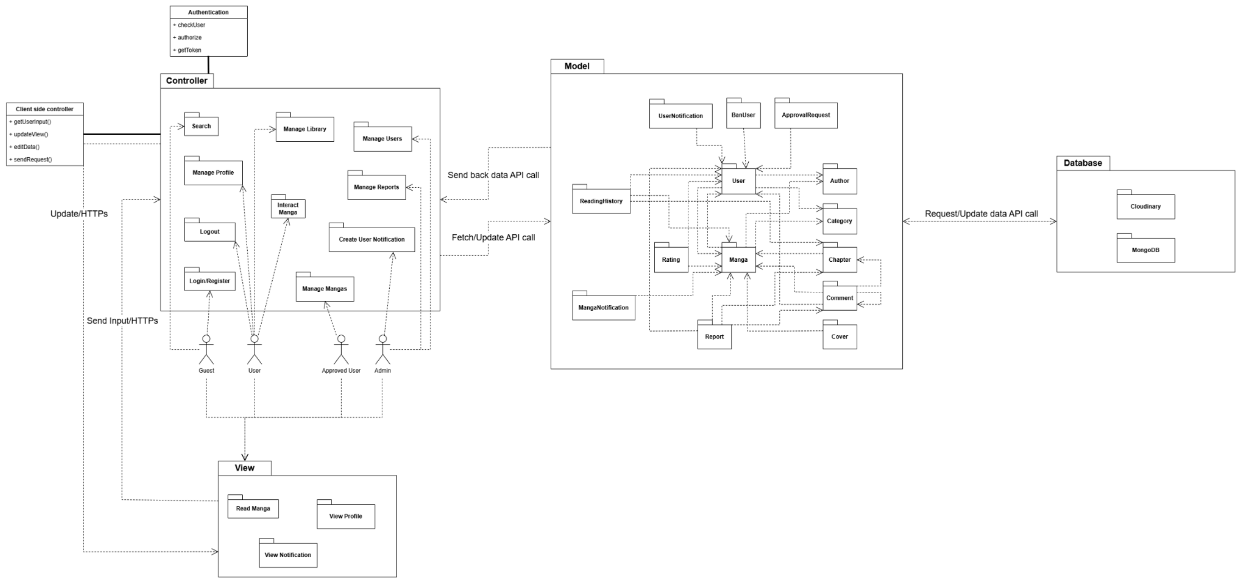
- Application environment: Web.
- Programming language and platform: Javascript (Node.js, ReactJS, Express.js), HTML, CSS.
- Database environment: MongoDB (this is NoSQL Database, for large scale and real-time applications).
- Security: password must be encrypted before storing in the database.
- Copyright requirement: Users must make sure that the posted manga are their own product.
- Responsive requirement: laptop, pc, smartphone, tablet…
- In addition, there are also constraints on the application as follows:
    - Simplicity
    - Scalability: includes some new features that are more suitable in practice (related to security, copyright, ...)

3. Use-Case Model



full use case model.png

4. Logical View



package diagram.png

1. Client-Server architecture

The client connects to the server and gets access to the services offered by the server.

In order to achieve this, the client sends requests to the server, and waits for the responses. The client then takes action based on the response such as displaying the received data, or redirecting to another page.

On the server side, the server listens for requests and processes them concurrently to achieve high efficiency. The server is responsible for sending back the data that the client requests, creating, updating or deleting the data in the database based on the requests.

2. Model-View-Controller architecture

A well-known software architecture pattern that separates an application into 3 main components:

- Model: responsible for application's data management.
- View: responsible for displaying the data to the users and enabling user interaction.
- Controller: responsible for receiving user input from the View, it then manipulates the Model to update data and based on that, it updates the View to reflect the change.

3. Technologies of choice

a. Client

ReactJS is the current most popular Front End library for web development. It is powerful, efficient and is used for building dynamic and interactive user interfaces, primarily for single-page-applications (SPAs). It has gained significant popularity due to its efficient rendering techniques, reusable components, and active community support.

Along with ReactJS is Vite, a build tool that offers excellent development speed by having an extremely fast Hot Module Replacement. Thanks to this, the server using Vite always starts up near instantly, typically less than 1 second, and the refresh time is also faster. The developers will have an easier time debugging and writing experimental code using Vite.

b. Server

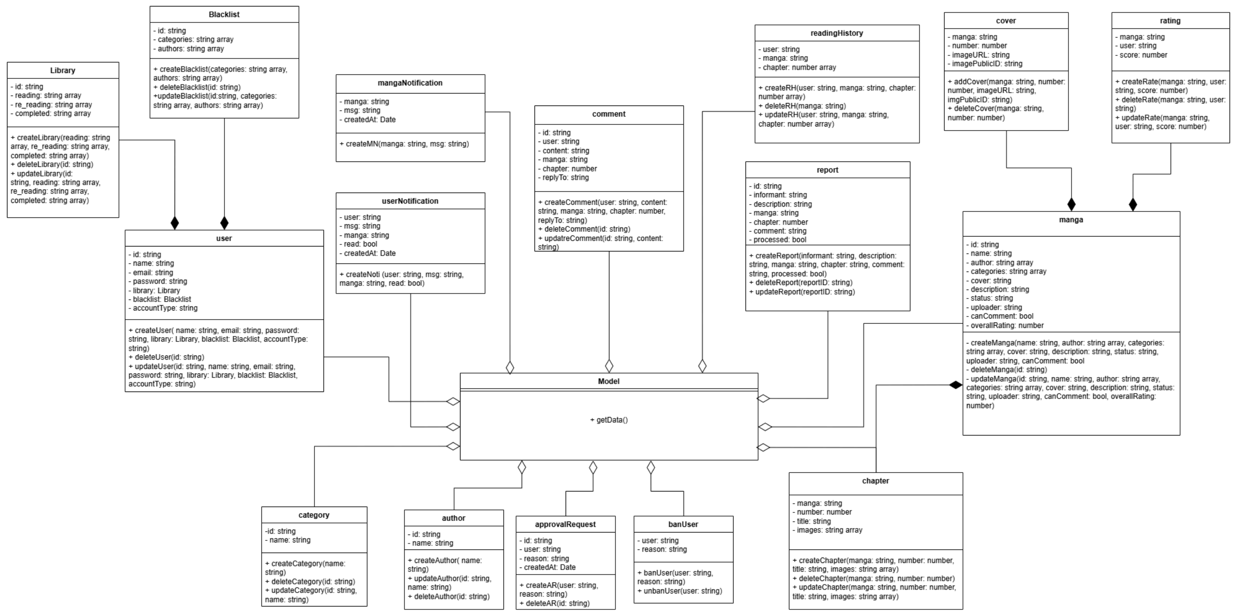
Thanks to NodeJS, Javascript can now run on the server side, allowing both the Back End and Front End to be written in the same programming language.

ExpressJS is one of the most popular Back End frameworks, it is a minimal and flexible framework that helps developers build Back End servers for CRUD applications easily. ExpressJS offers a robust set of features, making it easier to organize application's functionality with middleware and routing.

c. Database

MongoDB is a popular NoSQL database, offering flexibility and scalability.
Instead of the traditional tables and rows like in other SQL databases, MongoDB stores data in collections and documents. Each document is made up of key-value pairs similar to a JSON object, in this case a BSON object (developed by MongoDB themselves). This means that documents in a collection can have different structures, and some documents may be partially complete.
Additionally, Mongoose is a library that simplifies using MongoDB. It provides a feature to add structure to MongoDB collections with Schema validation while also keeping the flexibility of MongoDB documents. It is also built with helpful methods for developers to run complex queries.

4.1 Component: Model


Blacklist
- id: string
- categories: string array
- authors: string array
+ createBlacklist(categories: string array, authors: string array)
+ deleteBlacklist(id: string)
+updateBlacklist(id:string, categories: string array, authors: string array)
Library
- id: string
- reading: string array
- re_reading: string array
- completed: string array
+ createLibrary(reading: string array, re_reading: string array, completed: string array)
+ deleteLibrary(id: string)
+ updateLibrary(id: string, reading: string array, re_reading: string array, completed: string array)
mangaNotification
- manga: string
- msg: string
- createdAt: Date
+ createMN(manga: string, msg: string)
userNotification
- user: string
- msg: string
- manga: string
- read: bool
- createdAt: Date
+ createNoti (user: string, msg: string, manga: string, read: bool)
user
- id: string
- name: string
- email: string
- password: string
- library: Library
- blacklist: Blacklist
- accountType: string
+ createUser(name: string, email: string, password: string, library: Library, blacklist: Blacklist, accountType: string)
+ deleteUser(id: string)
+ updateUser(id: string, name: string, email: string, password: string, library: Library, blacklist: Blacklist, accountType: string)
comment
- id: string
- user: string
- content: string
- manga: string
- chapter: number
- replyTo: string
+ createComment(user: string, content: string, manga: string, chapter: number, replyTo: string)
+ deleteComment(id: string)
+ updatreComment(id: string, content: string)
readingHistory
- user: string
- manga: string
- chapter: number array
+ createRH(user: string, manga: string, chapter: number array)
+ deleteRH(manga: string)
+ updateRH(user: string, manga: string, chapter: number array)
cover
- manga: string
- number: number
- imageURL: string
- imagePublicID: string
+ addCover(manga: string, number: number, imageURL: string, imgPublicID: string)
+ deleteCover(manga: string, number: number)
rating
- manga: string
- user: string
- score: number
+ createRate(manga: string, user: string, score: number)
+ deleteRate(manga: string, user: string)
+ updateRate(manga: string, user: string, score: number)
report
- id: string
- informant: string
- description: string
- manga: string
- chapter: number
- comment: string
- processed: bool
+ createReport(informant: string, description: string, manga: string, chapter: string, comment: string, processed: bool)
+ deleteReport(reportID: string)
+ updateReport(reportID: string)
manga
- id: string
- name: string
- author: string array
- categories: string array
- cover: string
- description: string
- status: string
- uploader: string
- canComment: bool
- overallRating: number
- createManga(name: string, author: string array, categories: string array, cover: string, description: string, status: string, uploader: string, canComment: bool
- deleteManga(id: string)
- updateManga(id: string, name: string, author: string array, categories: string array, cover: string, description: string, status: string, uploader: string, canComment: bool, overallRating: number)
Model
+ getData()
category
-id: string
- name: string
+ createCategory(name: string)
+ deleteCategory(id: string)
+ updateCategory(id: string, name: string)
author
- id: string
- name: string
+ createAuthor(name: string)
+ updateAuthor(id: string, name: string)
+ deleteAuthor(id: string)
approvalRequest
- id: string
- user: string
- reason: string
- createdAt: Date
+ createAR(user: string, reason: string)
+ deleteAR(id: string)
banUser
- user: string
- reason: string
+ banUser(user: string, reason: string)
+ unbanUser(user: string)
chapter
- manga: string
- number: number
- title: string
- images: string array
+ createChapter(manga: string, number: number, title: string, images: string array)
+ deleteChapter(manga: string, number: number)
+ updateChapter(manga: string, number: number, title: string, images: string array)

modelComponent.drawio.png

The model component represents the structure of the data and its constraint. It is built to manage the data connected to our database (MongoDB) easily. All the functions to manage the database like create, update and delete are considered in the model component.

Important class attributes explanation:

- User class:
  - id: a string which is actually an objectID in MongoDB.
  - name: a string representing the full name of a user.
  - password: a string of a hashed password with salt.
  - library: a library object containing mangaId divided in three types of completion: reading, re_reading and completed
  - blacklist: a blacklist object containing a categories and authors list comprising of the corresponding id.
  - accountType: there are three account types: user, approved (approved user) and admin. When creating a new account, your account type will be 'user'.

- Manga class:
  - id: a string which is actually an objectID in MongoDB.
  - name: a string representing the full name of manga.
  - authors: a list containing all the authors of the manga.
  - categories: a list containing all the categories of the manga.
  - cover: a string which is the id of the cover.
  - description: a full description of the manga.
  - status: there are three status of a manga: in progress, completed and suspended.
  - uploader: a string which is the id of the user who uploads the manga.
  - canComment: a boolean to set if the manga can be commented by the users or not.
  - overallRating: an average score of all the users who have rated the manga.

4.2 Component: View

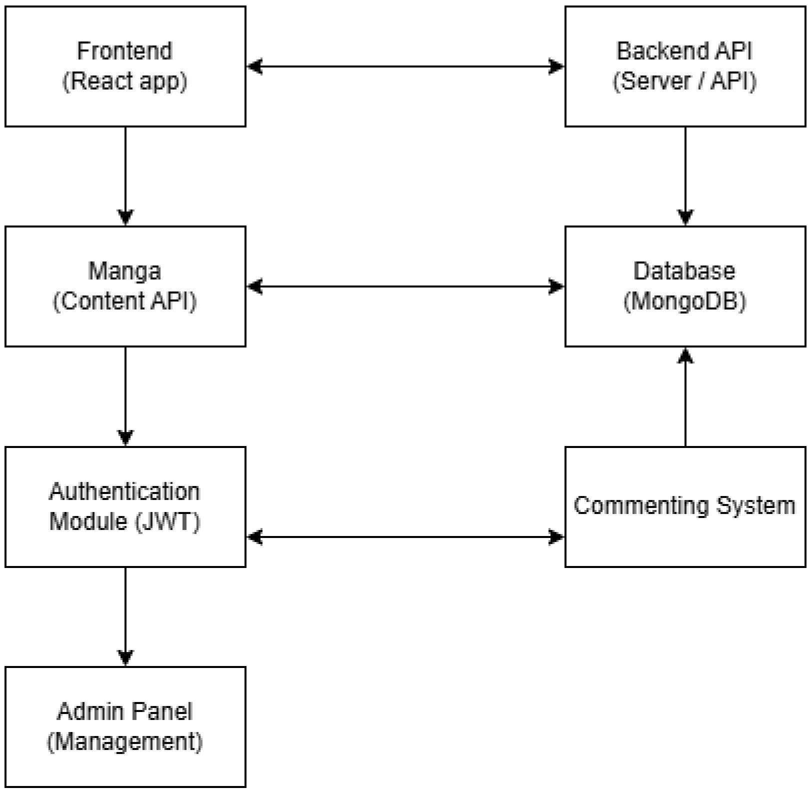
The view component represents how the data is fetched and viewed on the users' devices. It is responsible for writing and reading requests to other logical components to display the data the users want, for example, their feed or all of the names of authors we have stolen from.

These component templates are written using React, which is a library for writing HTML nodes within JavaScript, through the use of createElement(), for example, a <div> would mean calling React.createElement('div'). But this will get clunky really quickly, therefore, we use an extension of JS, essentially allowing HTML within JS, called JSX.

To make sure that the view component is robust, we try to integrate Browser Testing with Playwright to ensure automated tests can see and interact with the view the same way as any user would. But because of the extremely tight coupling between backend and frontend, this has proven to be very difficult.

Relationships:

- Components interact with controllers via sending HTTP requests, usually through the axios middleware. Activated by the user agent interacting with interactive elements, such as inputs, buttons and anchors.
- Components interact with models via receiving HTTP requests, by fetching, usually through the axios middleware, or wrapped with SWR. Activated by the user agent visiting a route, as a side effect of interactivity, or simply the "revalidation" part of 'Stale-While-Revalidation' technique.
- Components interact with each other via stores. Basically stateful values that exist in another file, and every component that imports such stores have access to the same data, and can influence and affect each other all at once.
- Components are displayed on the browser via the Vite/Rollup build tool.



Frontend (React App):

- Provides the user interface for browsing, reading manga, logging in, registering, and commenting.
- Interacts with the **Backend API** to fetch data.

Backend (API/Server):

- Handles business logic and provides APIs for the frontend to retrieve manga, chapters, comments, and manage users.
- Interacts with the **Database** to store and retrieve data.

Database:

- Stores manga data, chapters, user accounts, and comments.
- Could be MongoDB or MySQL.

Authentication Module:

- Manages user login and registration, ensuring secure authentication using techniques like **JWT**.

Commenting System:

- Allows users to comment on manga chapters.
- Interacts with the **Database** to store and retrieve comments.

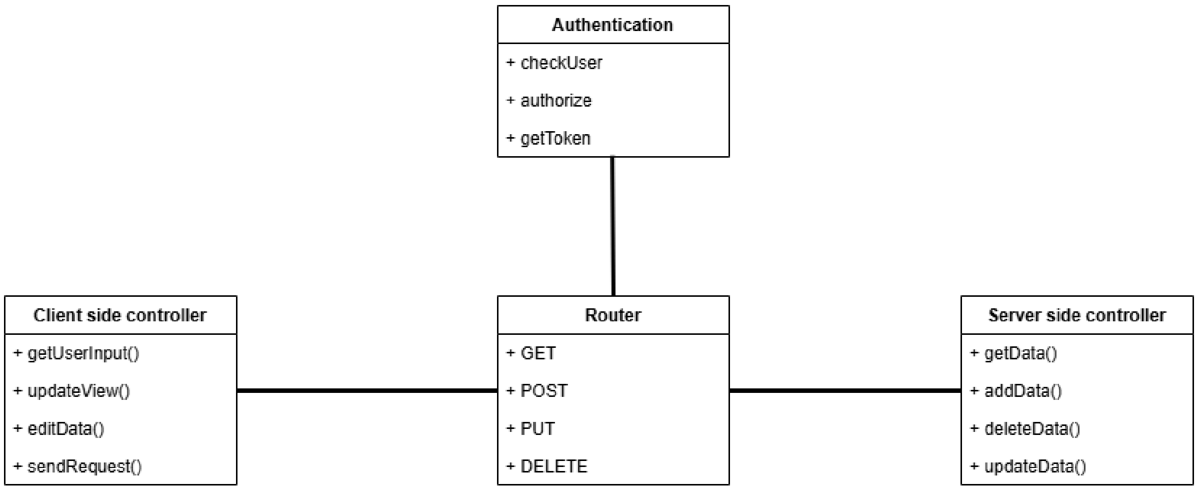
Manga Content API:

- Provides detailed information about manga (title, author, genre) to the frontend.

Admin Panel:

- Allows admin users to manage manga content, user accounts, and moderate comments.
- Interacts with the **Backend** to perform administrative actions.

4.3 Component: Controller



controller component.png

The controller component has the job of acting on the database via the model component's schemas. It is responsible for taking data sent as forms or bodies from the View component and acting on said data, modifying the state of the component Model, so the next time the View component queries on such state, it can read the newly updated values.

Types of controllers:

- Client-side controllers: input handlers that do light validation, bundle up forms and interactive data to send it to the backend server, usually through Axios or Fetch. These mainly reside on the browser and the frontend.
- Server-side controllers: the *main player* of this component. These are written as asynchronous handlers, fit in a sequence of other handlers called middlewares (for example, middlewares that intercept requests and block unauthorized ones, or middlewares that handle errors so that the server pipeline doesn't crash if unexpected errors happen). These are wired through Express Routing and hooked up the HTTP verbs (GET, POST, PUT, DELETE and HEAD).

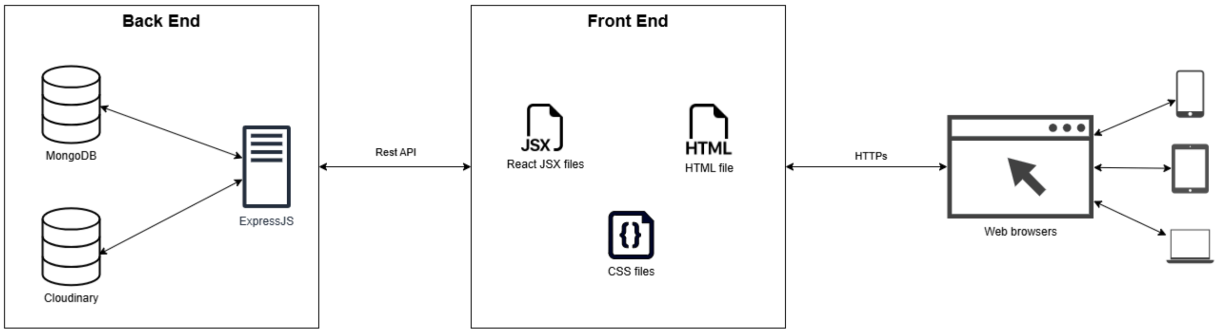
Relationships:

- Client-side controllers interact with the "View" component by taking data and requests from the user on the frontend and sending it to the backend controllers.
- Server-side controllers interact with the "Model" to do CRUD operations (Create, Read, Update, Delete) on databases, whereas client-side controllers can't interact directly with the Model, but

needs a server-side controller handling that request.

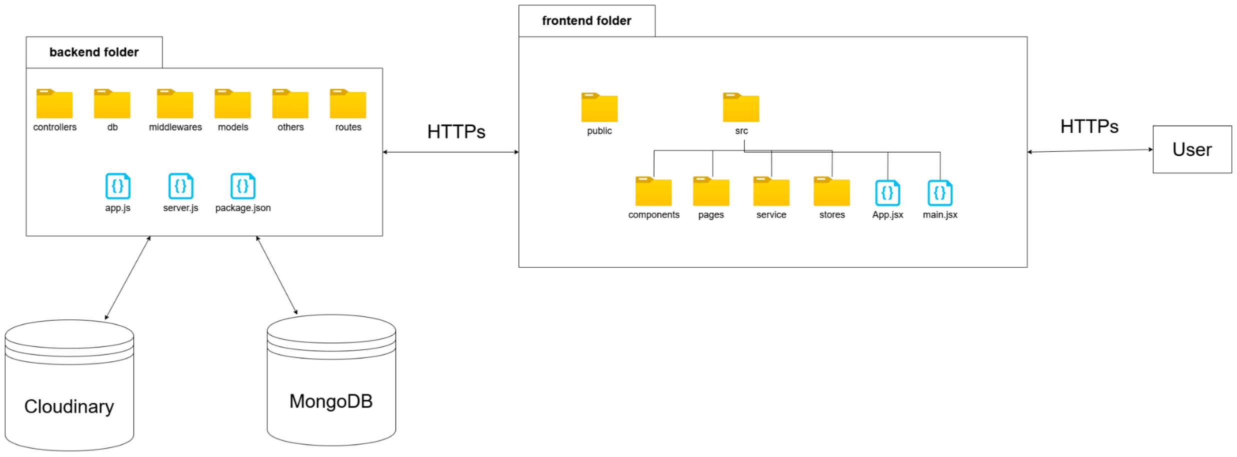
Coupling: controllers are meant to be one-man, one-job. One controller will have its own specific job to accomplish and has to accomplish it well. There shouldn't be weird coupling of other controllers to prevent unpredictable results.

5. Deployment



Our website intends to serve as many devices as possible, ranging from big screens like PCs to small screens like Mobile Phones, as well as be compatible with a wide range of web browsers like Chrome, Firefox, Safari… But since our time is limited, we will prioritize PCs and mobile phones first.

6. Implementation View



The **frontend folder** includes:

- **public**: folder that contains images for decorating in our pages, make public to the users.
- **src**: containing the actual code of the frontend.
  - **components**: the folder that contains our individual components for pages.
  - **pages**: the folder containing JSX files that assembles other components to make up pages.
  - **service**: this folder contains API calls to the server to fetch resources as well as modifying the database.
  - **stores**: contains files that export global states using the package nanostores (a tiny framework-agnostic state manager, using the React integration).

- ○ **App.jsx and main.jsx**: entries point of the frontend code, automatically generated by the command. We added the routing inside App.jsx.

As for our **backend folder**:

- **app.js**: contains code that initializes our Express server. This also contains code that routes requests to different route files.
- **server.js**: imports app.js and starts the server. We do this in order to use Vitest for testing.
- **db**: contains a single db.js to initiate connection to the MongoDB server.
- **routes**: contains js files that route each request to their corresponding controller function.
- **controllers**: contains js files, each of which contains logic to handle requests sent by the users.
- **models**: contains js files of mongoose models.
- **middlewares**: contains our custom middlewares to check for authentication as well as handle errors.
- **others**: contains cloudinaryWrapper.js, which is used to handle images uploading/deleting on Cloudinary.