BookLeaf

Software Architecture Document

Version <2.0>

Revision History

| **Date** | **Version** | **Description** | **Author** |
| --- | --- | --- | --- |
| 20/11/2024 | 1.0 | Section 1, 2, 3. | Pham Tran Yen Quyen, Pham Thanh Vinh |
| Section 4: diagrams | Do Dinh Hai |
| Section 4: descriptions | Truong Thanh Toan |
| 28/11/2024 | 2.0 | Section 4: Add analytics table for the erd at database  Section 5: diagram and descriptions  Section 6: diagram and descriptions | Do Dinh Hai |
| Section 4: Reconstruct general logic view for the system.  Section 4: Break down the MVC models into more specific sections that are suitable for the general logic view. | Pham Thanh Vinh |

Table of Contents

[**1. Introduction 4**](#_heading=h.gjdgxs)

[1.1 Purpose 4](#_heading=h.9lzqv8q2oyka)

[1.2 Scope 4](#_heading=h.abmai2dfehvg)

[1.3 Definitions, Acronyms and Abbreviations 4](#_heading=h.z96bf73zo62w)

[1.4 References 4](#_heading=h.6jm2bsjp7wb7)

[1.5 Overview 4](#_heading=h.hxptkao8gfol)

[**2. Architectural Goals and Constraints 4**](#_heading=h.30j0zll)

[2.1 Architectural Goals 4](#_heading=h.alwugdhb1ce0)

[2.2 Constraints 5](#_heading=h.pab3oc1ay00m)

[**3. Use-Case Model 5**](#_heading=h.1fob9te)

[**4. Logical View 10**](#_heading=h.3znysh7)

[4.1 Component: Backend 11](#_heading=h.vnwrympid4ym)

[4.1.1 Component: Controller 11](#_heading=h.e58vg5uq9i7t)

[4.1.2 Component: Model 12](#_heading=h.jmqms06f6s7b)

[4.1.3 Component: Router 13](#_heading=h.g6id8mo2a2nm)

[4.2 Component: Database 14](#_heading=h.pxzt1rfwo933)

[4.3 Component: Frontend Application 15](#_heading=h.2et92p0)

[**5. Deployment 19**](#_heading=h.tyjcwt)

[**6. Implementation View 20**](#_heading=h.3dy6vkm)

Software Architecture Document

# Introduction

## Purpose

This Software Architecture Document (SAD) provides a comprehensive architectural overview of the BookLeaf system. It is intended to serve as the primary technical reference for the development team and course instructor, capturing and conveying the significant architectural decisions that have been made in the system's development.

## Scope

The document covers the software architecture of BookLeaf, a web-based e-library platform utilizing the MERN (MongoDB, Express.js, React.js, Node.js) technology stack. The architecture supports core e-library functionalities while operating under zero-budget constraints.

## Definitions, Acronyms and Abbreviations

**MERN**: MongoDB, Express.js, React.js, Node.js - Full-stack JavaScript framework.

**API**: Application Programming Interface.

**REST**: Representational State Transfer.

**JWT**: JSON Web Token.

**SPA**: Single Page Application.

**BookLeaf**: Web-based e-library platform for reading and sharing digital books.

## References

* BookLeaf Vision Document v2.0
* Software Development Plan v2.0
* Use-case Specification v1.0

## Overview

This Software Architecture Document is organized into the following sections:

* Section 2 - Architectural Goals and Constraints: Describes the key requirements, constraints, and objectives that significantly influence the architecture, including security, performance, and development constraints.
* Section 3: Use-Case Model - Presents the use case diagrams, showing the system's main functional requirements.
* Section 4: Logical View - Details the system's architectural components and their relationships
* Section 5: Deployment - Explains how the system components are deployed across different machines and environments.
* Section 6: Implementation View - Provides the folder structures and organization of the codebase.

# Architectural Goals and Constraints

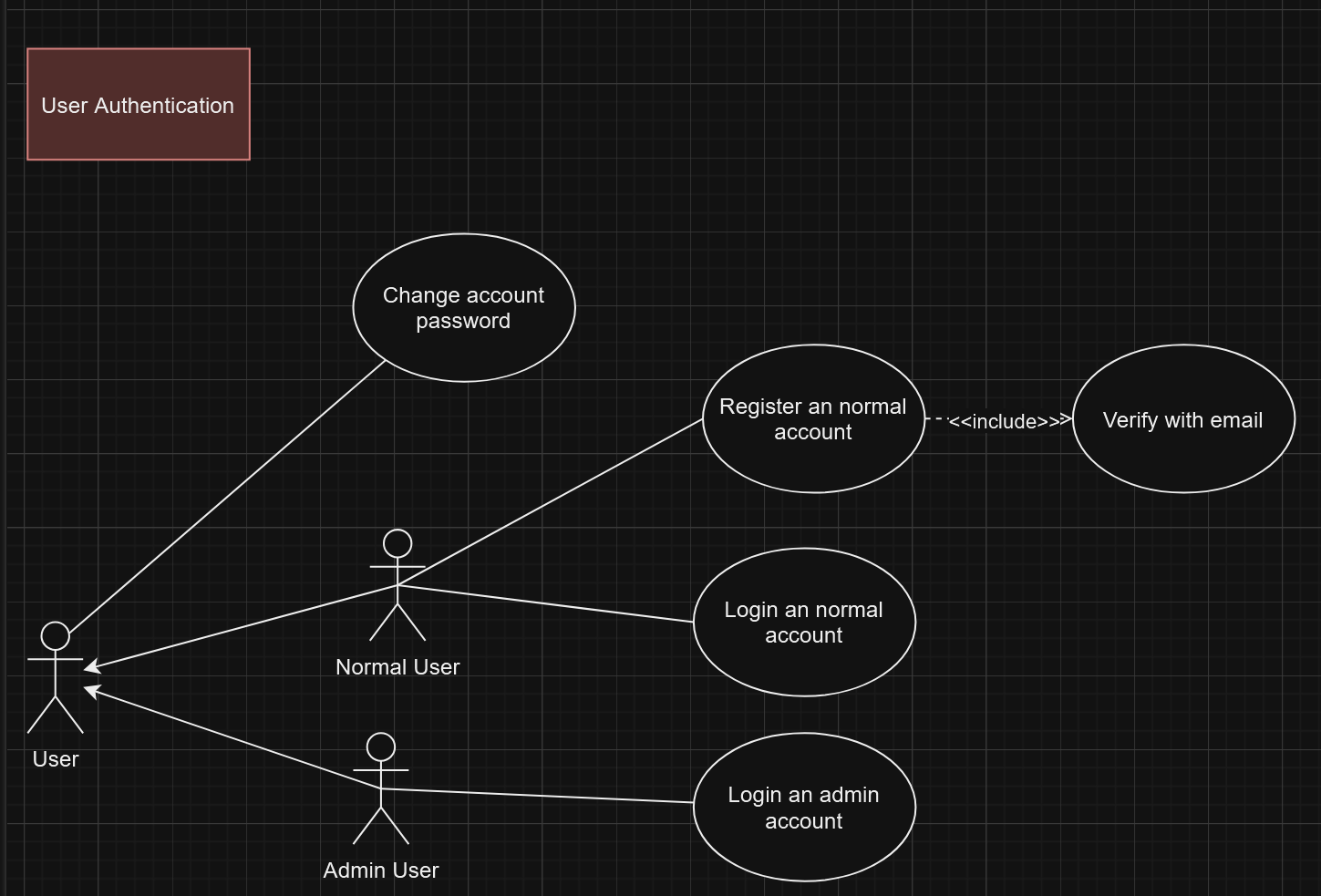
## Architectural Goals

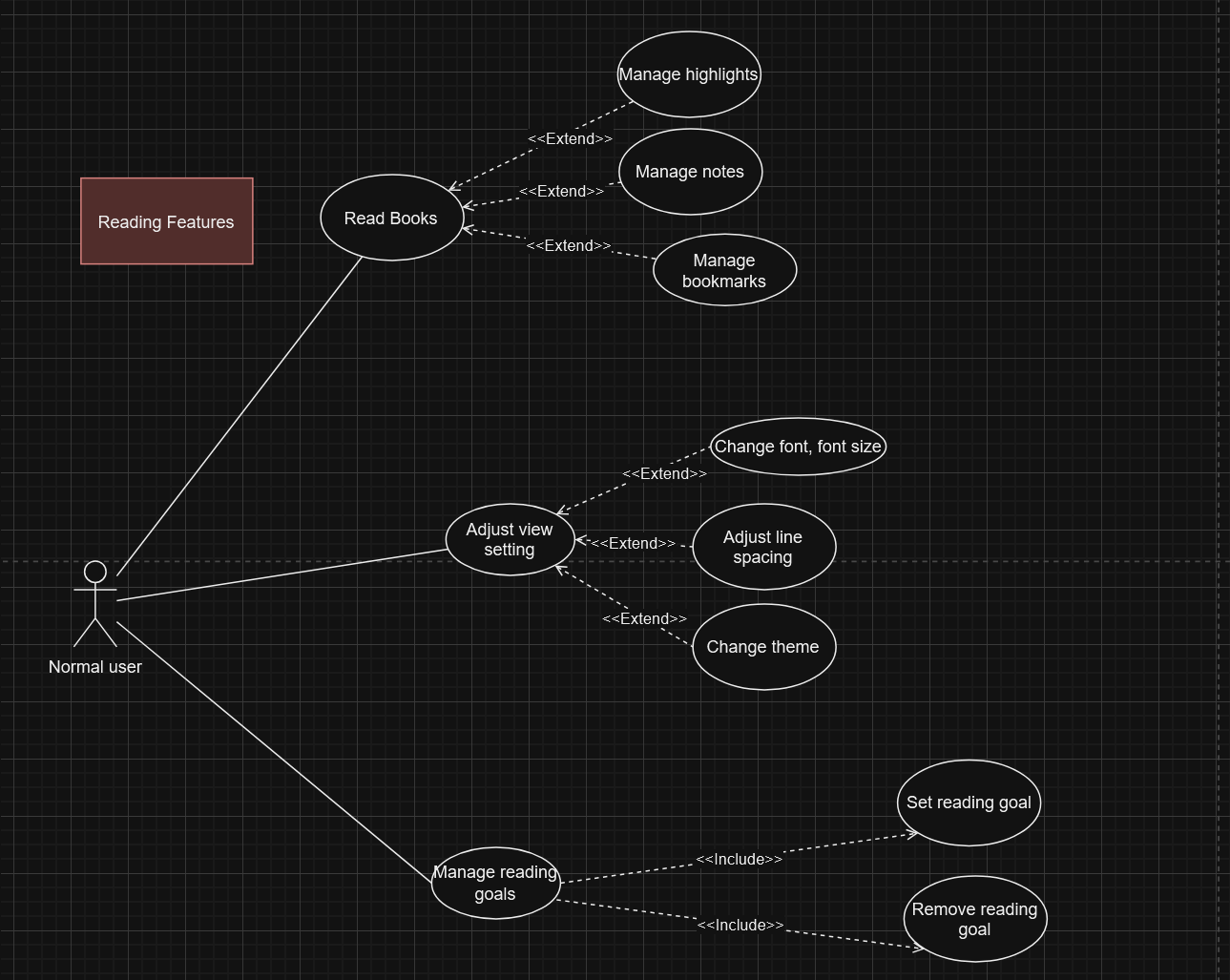
* Performance Goals:
* Page load time must be under 3 seconds under normal conditions.
* Document rendering must complete within 4 seconds for standard PDF/EPUB files.
* Scalability Goals:
* Efficient handling of increasing document storage.
* Support for a growing user base.
* Security Goals:
* Secure user authentication and authorization.
* Protected storage of user data and uploaded files.

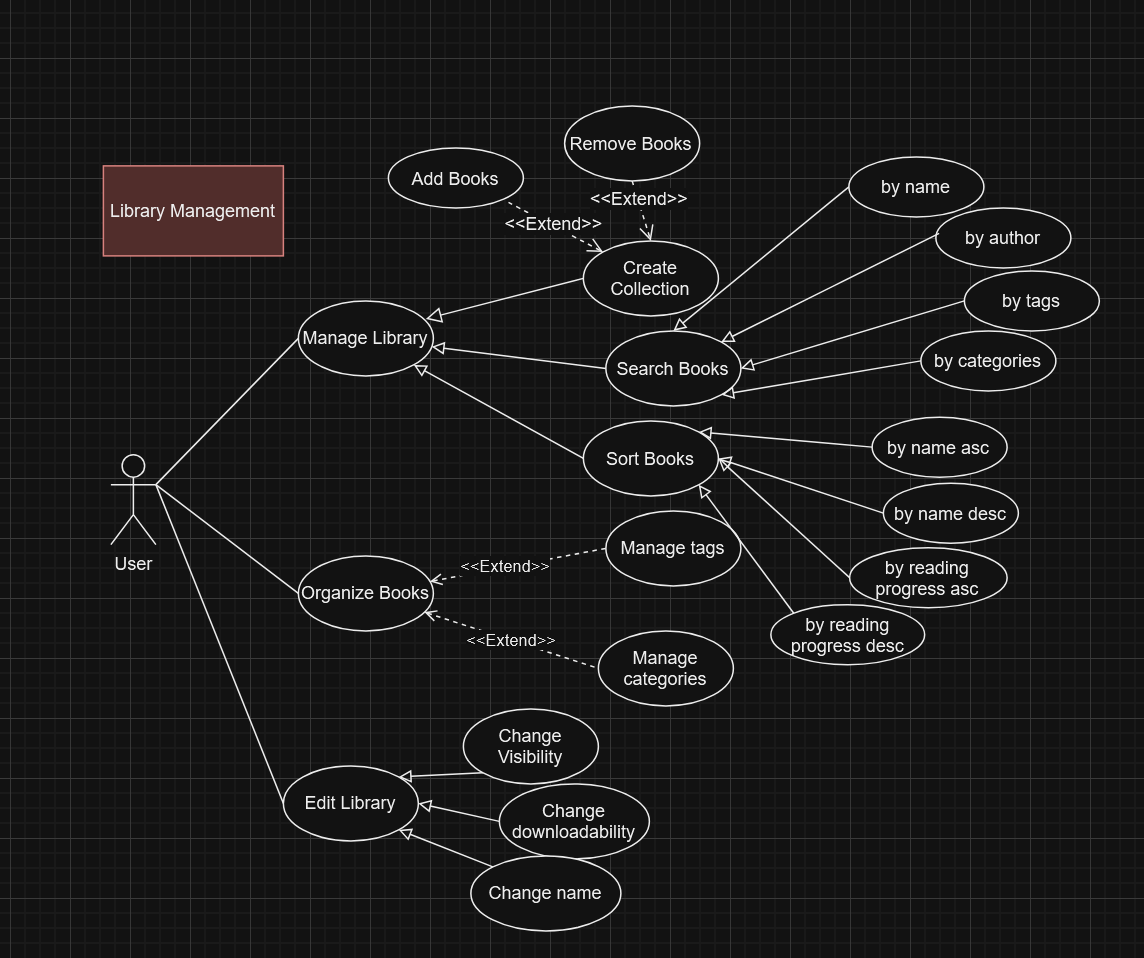
## Constraints

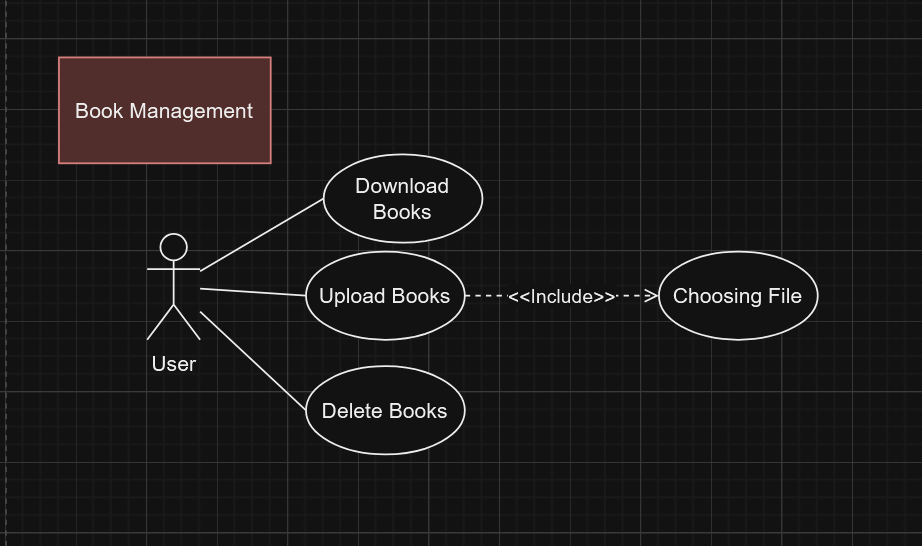
* Technical Constraints:
* Zero-budget requirement limiting technology choices to free tiers.
* Web-based platform constraints: Must support the latest two versions of major browsers, must be responsive for different screen sizes.
* Development Constraints:
* Fixed 10-week development schedule.
* Four-member development team with shared responsibilities.
* Implementation Strategy: Using MERN stack
* MongoDB for document-based data storage
* Express.js for backend API development
* React.js for frontend SPA development
* Node.js for server runtime environment

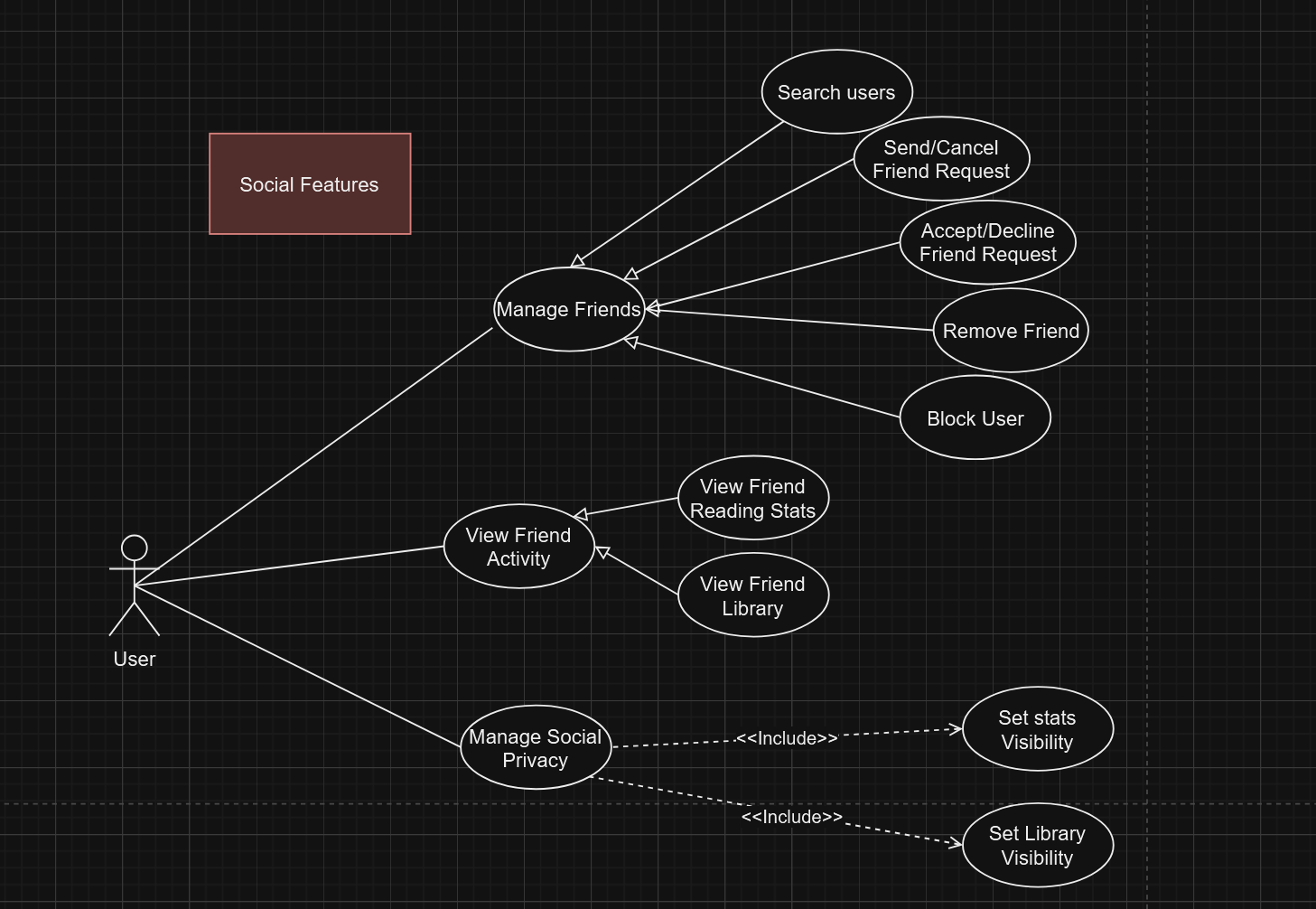
# Use-Case Model

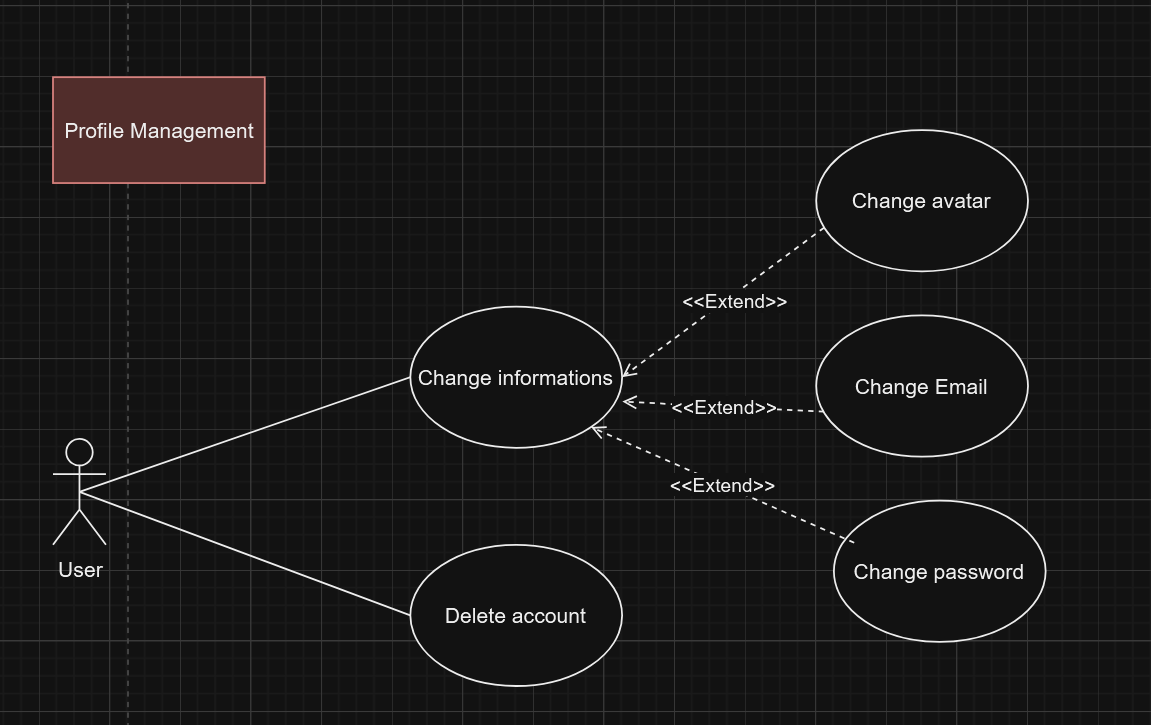








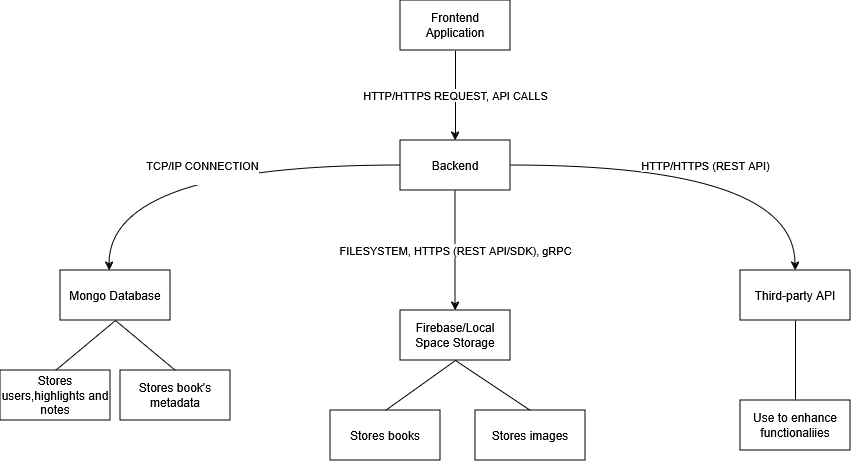




# 

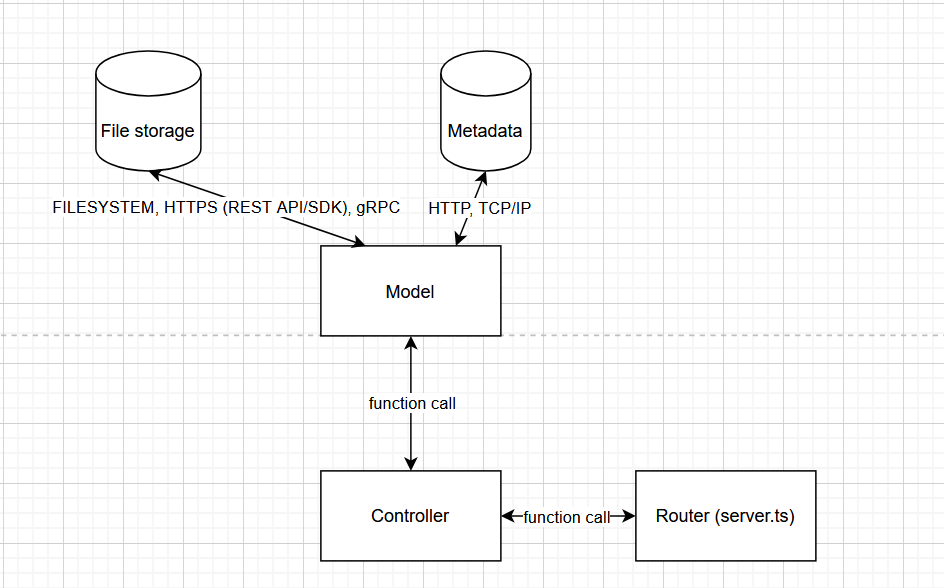
# Logical View

This is our overall components logic view.



## Component: Backend

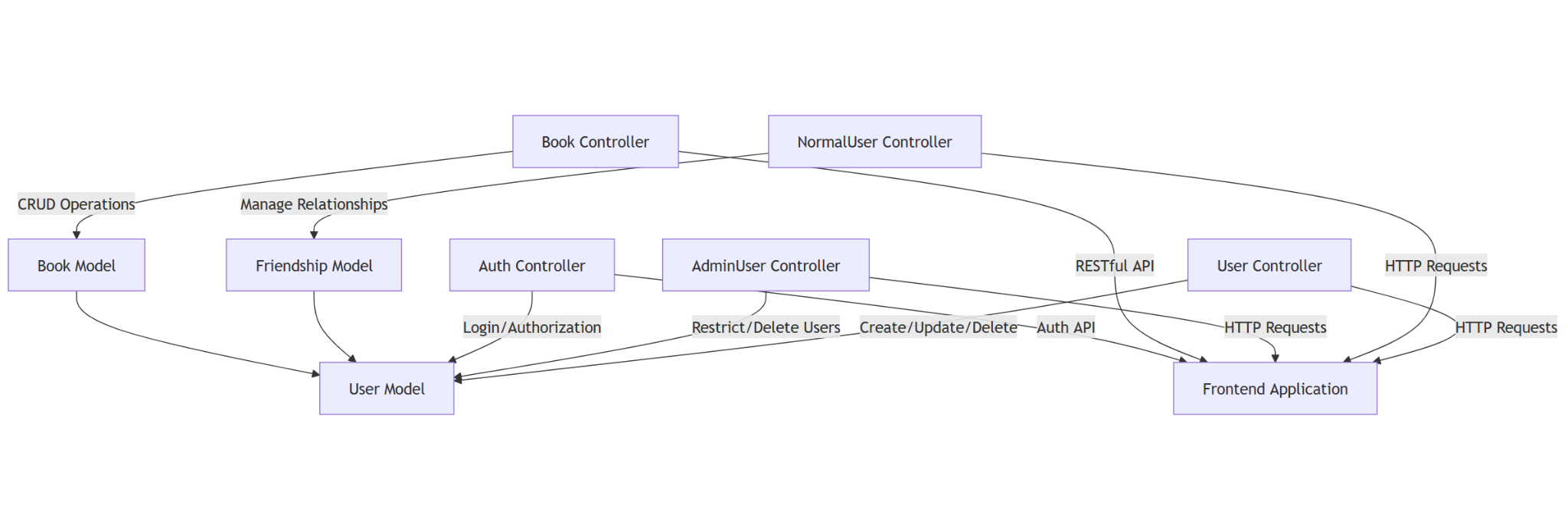
We use the MVC model as below for Backend.



* Brief description:
  + Router catches the url and routes the request to a specific controller.
  + Controller handles the request (logical handling, CRUD on the database via Model component if essential) to obtain data for View.
  + The Model component enforces the Controller to CRUD on the database following business rules.
  + Data obtained for View will be sent back to the browser for rendering.
* More details will be given in the following subsections.

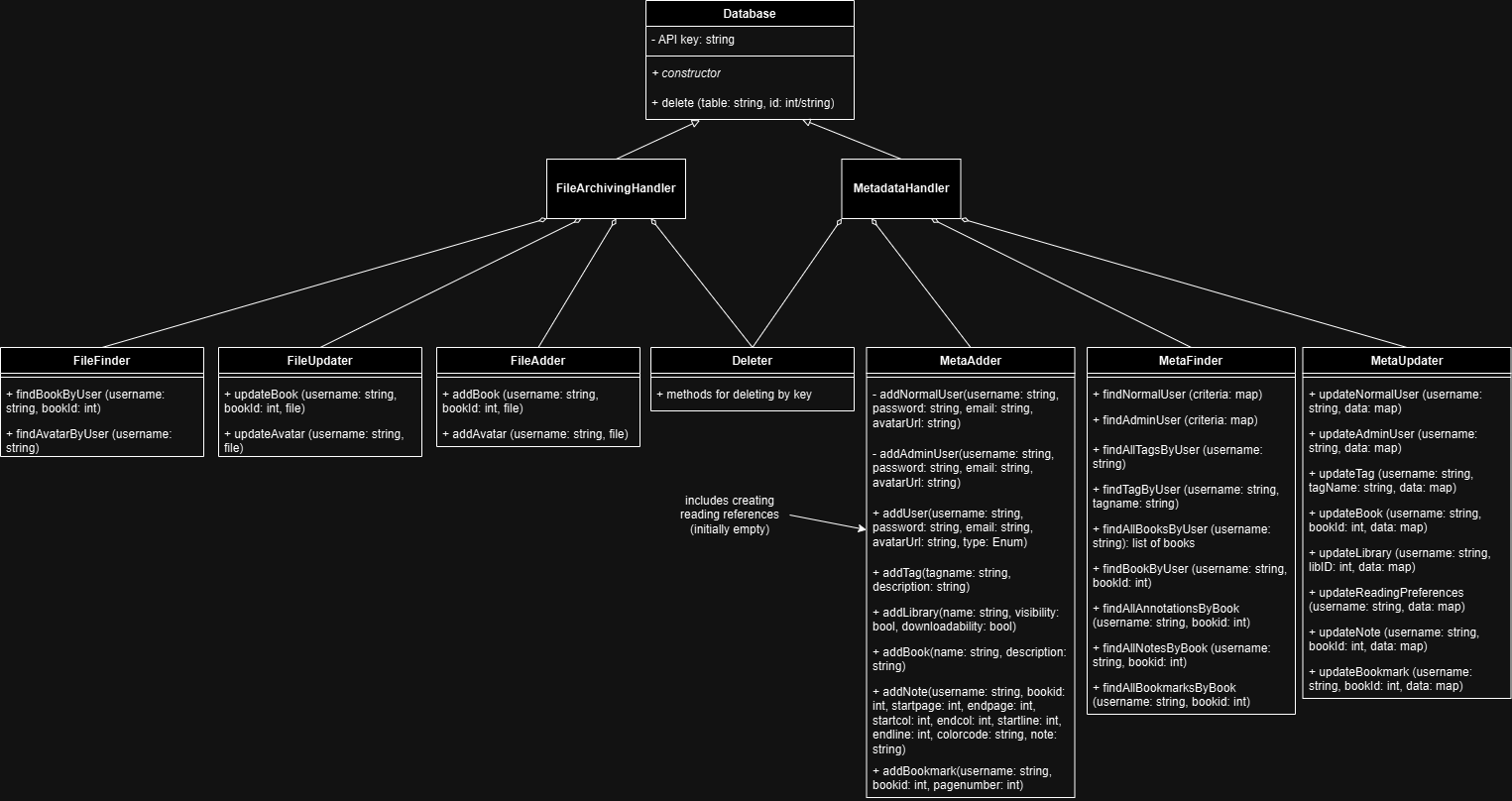
## Component: Controller

* All components in the server will be programmed using JS, hosted using ExpressJS and run on a NodeJS environment.
* Each component will communicate with each other using environment variables (either global or local), HTTP requests and middlewares.
* Components in the server (controllers) will be connected to UI (views) using HTTP requests (Restful API).
* When requests from the frontend application, the controller will delegate tasks to the services to handle it.
* ***User***: controller for creating, changing attributes of a user, such as username, email, password, etc.
  + ***NormalUser***: invoking related classes’ methods, namely Friendship, BlockingRelationship and FriendRequest; possesses ***Book***s, ***Tag***s, list of friends, list of blocked users.
  + ***AdminUser***: monitoring normal users, this includes restricting, deleting accounts or getting an analytical view on all users.
* ***ReadingPreferences***: managing a user’s reading preferences, this includes creating, updating, reading and deleting the user’s preferences. It belongs to a ***Normal User***.
* ***Book***: controller for managing a user’s uploaded books, managing a book’s tag, as well as serving the book to view; possesses a list of annotations.
* ***Tag***: controller for managing user-defined tags, overseeing their attributes and life cycle.
* ***Annotation***: superclass controller who manages annotation of books.
  + ***Note***: subclass manages document noting, as well as their attributes (color, content).
  + ***Bookmark***: manages page bookmarking, and their attributes (page number).
* ***Friendship***: manages relationships of a user, as well as removing the friend from the user’s friend list.
* ***FriendRequest***: manages requests between 2 users, a sender and a receiver, subsequently overseeing the receiver action after receiving the request.
* ***BlockingRelationship***: manages the one-way action between 2 users, a blocker and a blocked, restricting options in the future for the blocked on the blocker.
* ***Authenticator***: manager of all process related to authentication and authorization
  + ***AdminAuthenticator:*** a subclass of authenticator, granting higher level of permissions and services.
  + ***NormalUserAuthenticator***: run-off-the-mill authorization and authentication process, granting normal user ordinary services and permission.
* ***SyntaxChecker***: depending on the authentication process, syntax checker is for checking, validating syntax of username, email, password, or hashing, validating password, etc.
* ***SingletonEmailVerifier***: a singleton object to oversee the verifying process of email, as well as send verification code.



## Component: Model

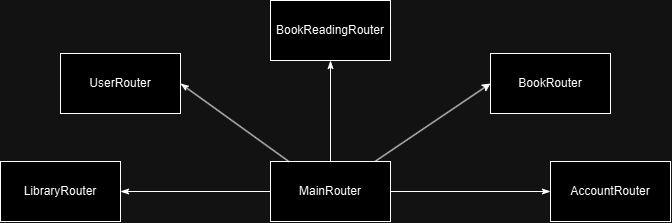
* This interface assists the controller to interact with the database.
* This component will be programming using JS and run on a NodeJS environment.



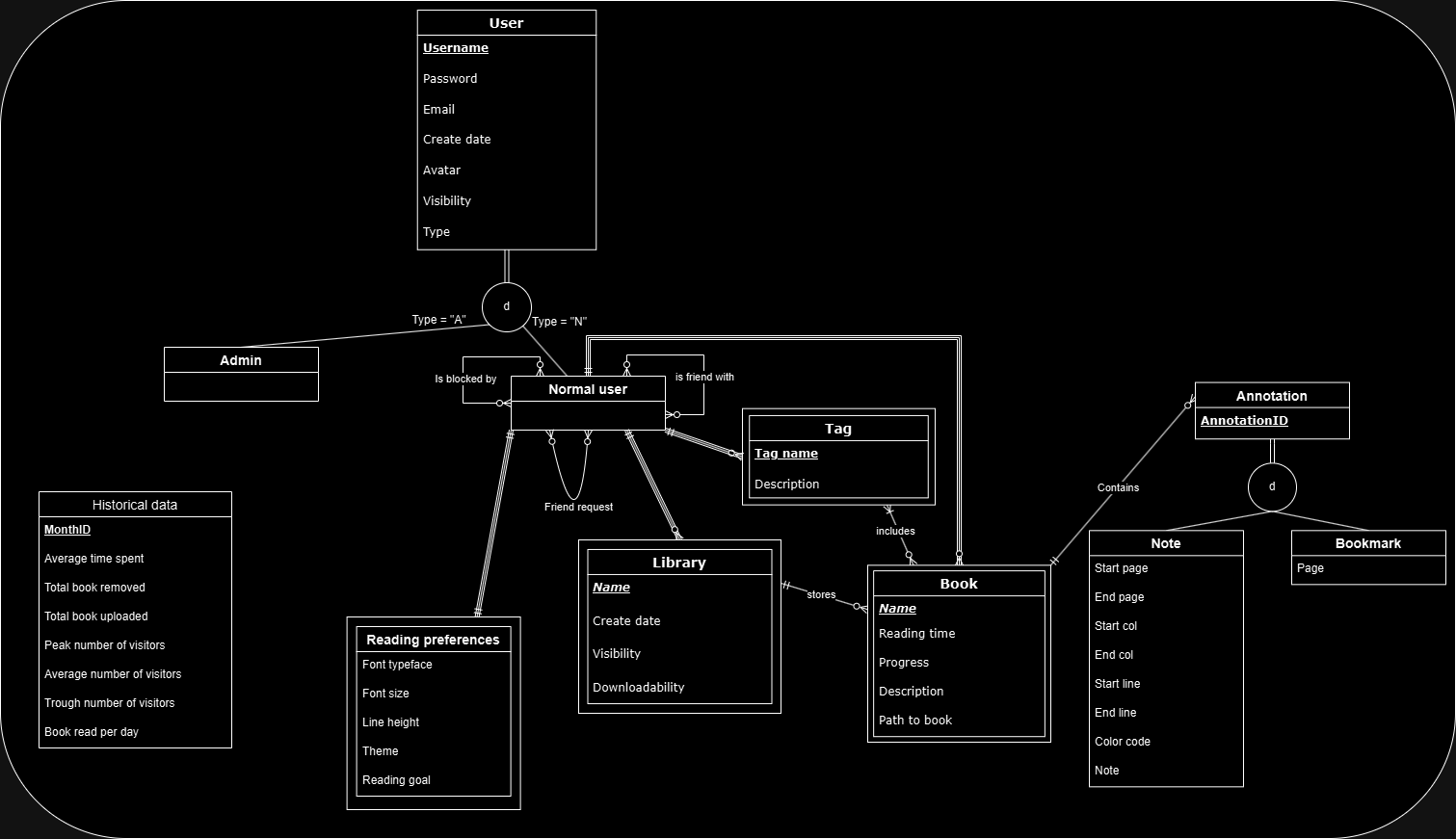
* Description:
  + **Adder**: for adding data to the database.
  + **Finder**: for finding data in the database.
  + **Deleter**: for deleting data from the database by key. It means that the programmer should invoke the Finder first to obtain the Id, and then the Deleter to delete. The keys are:
    - *User*: username.
    - *Book*: username, bookName.
    - *Tag*: username, tagName.
    - *Library*: username, libraryName.
    - *Note*: username, bookName, annotationId.
    - *Bookmark*: username, bookName, annotationId.
  + **Updater**: for updating data in the database.
  + There are 2 databases responsible for 2 different tasks.
    - ***Metadata***: includes SINGLETON MetaAdder, MetaFinder, and MetaUpdater.
    - ***Files***: includes SINGLETON FileAdder, FileFinder, and FileUpdater.
    - They share a common Deleter.
  + Note: each instance of *Reading Preferences* is attached to a particular user.

## Component: Router

* Description
  + **Main Router**: is the middleman for all other routers, acting like an overseer, watching over all incoming and outgoing requests and responses and handling them to the correct router for data processing, analysis, etc.
  + **AccountRouter**: is the router responsible for account management, specifically creating, logging in and changing password or password recovery.
  + **BookReadingRouter**: is the handler responsible for serving books and its metadata to the view. BookRouter: is responsible for querying a database to perform CRUD operations, such as changing tags, uploading, downloading, removing books and their metadata, as well as analytical data for books.
  + **UserRouter**: is responsible for creating routes for users and their data displaying, routing to the user site. This consists of retrieving, updating, deleting information related to users.
  + **LibraryRouter**: is responsible for managing libraries, this includes creating, removing, changing attributes about a library.



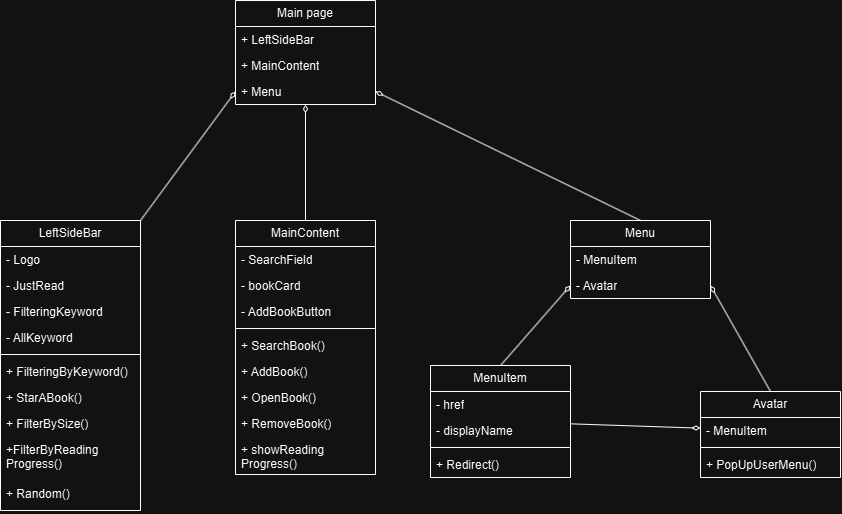
## Component: Database



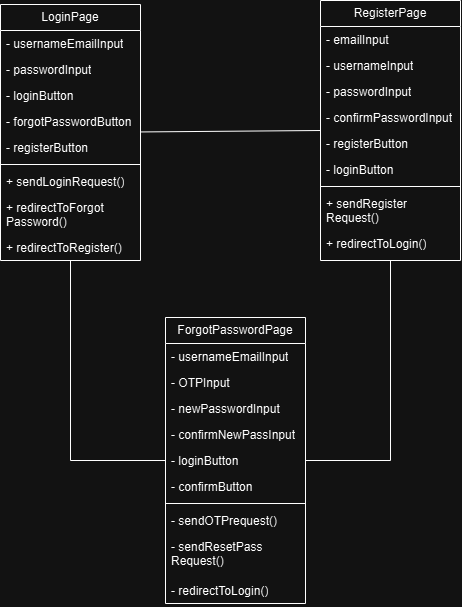
* All components in the database will be constructed through either native methods in MongoDB (for metadata) and Firebase (for users’ avatar and book files), or be programmed and seeded using JS methods and middleware in the server.
* All components in the database will communicate with the server by using Object Data Modeling (ODM) like Mongoose or a combination of Software Development Kit (SDK) and HTTP requests like Firebase/Firestore kit in JS/NodeJS.
* Description:
  + ***User***: a user in general.
    - There are 2 types of user: Admin user and Normal user.
    - Admin user: there is nothing different between an admin user and a general user in terms of attributes.
    - Normal user: a normal user is the book reader, who has their own reading preferences, library, collection of books they uploaded,...
  + ***Reading preferences***: one normal user has only one set of reading preferences, including font face, font size, line spacing,...
  + ***Library***: one normal user can have at least 0 and at most 20 libraries.
  + ***Book***: metadata of books uploaded by an individual normal user.
  + ***Tag***: a set of tags created by an individual user; one normal user can have at least 0 and at most 20 tags.
  + ***Annotation***: a set of annotations related to a specific book.
    - ***Note***: a note marked by a highlight.
    - ***Bookmark***: user can create a bookmark with name.
  + **Historical data**: store the analytical data of the whole system each month, provide admin with an option to draw charts, graphs and get an analytical view of the system. Only the server and the admin have access to this data. This is crucial for decision making processes and strategies of the company in the upcoming months.
* Note:
  + ***Friend request***, ***Blocklist***, ***Friendlist*** (which are relationships between 2 normal users) will form 3 stand-alone collections / tables when being implemented.
  + ***Reading preferences*** and ***User*** information will be stored in the same collection when implementing.
  + ***Historical data*** is independent of any other entity.

## Component: Frontend Application

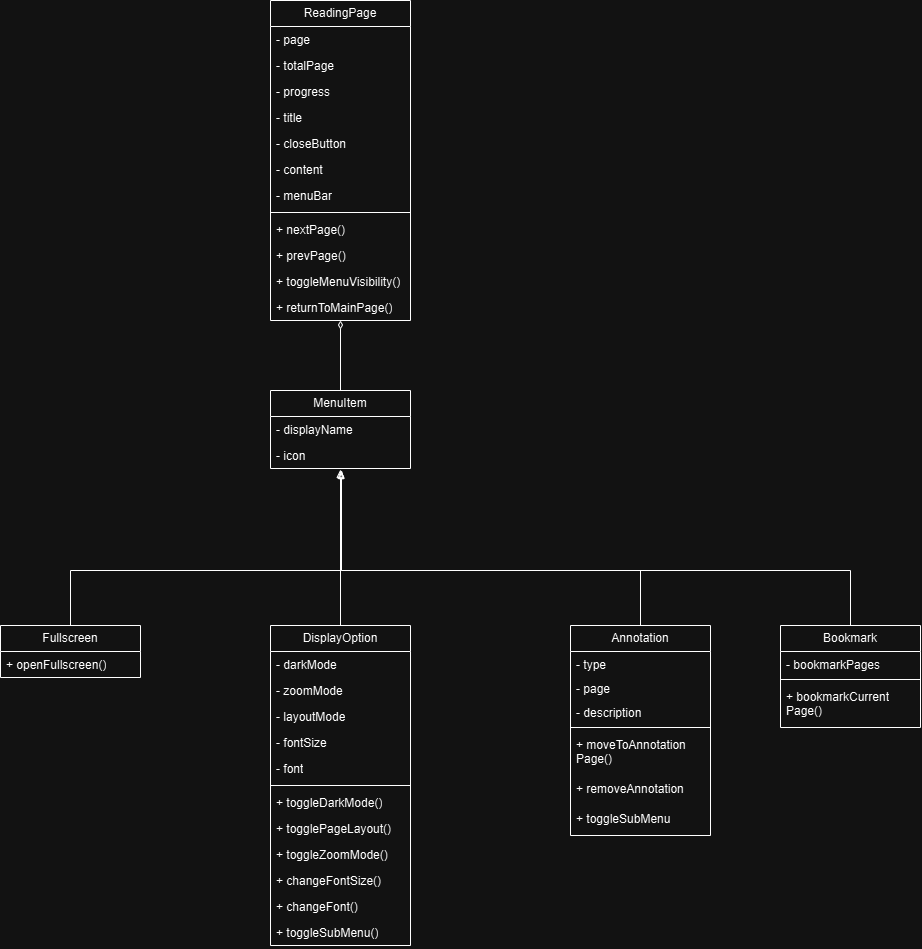
* All components in UI/UX will be programmed using ReactJS and hosted with Vite. Therefore, the languages used for building the application will be TSX, CSS, and HTML.
* The UI components are connected with each other using either HTTP requests (GET/POST requests) or the ReactJS importing method, which imports the whole component as a partial.
* The UI components in views communicate with the controllers in the server using HTTP requests (Restful API), which follow the CRUD (Create, Read, Update, and Delete) process.
* **Main page**
  + The main page component is the container that holds other sub-components. The main page requires users to sign in to access its content and functionalities.
  + LeftSideBar is used to filter contents, quickly access recently read books, or open a random book from the user’s library.
  + MainContent consists of the user’s library, from which the user could access their books, and the search bar for filtering books.
  + The menu consists of menu items, which will redirect the user to other pages/functionalities, within the item named user, represented as the user’s avatar, will be another sub-menu with account setting options. The user menu will have menu items that affect the user's settings and experiences.
  + Main page will use the combination of GET and POST requests, based on the need of security and accessibility.



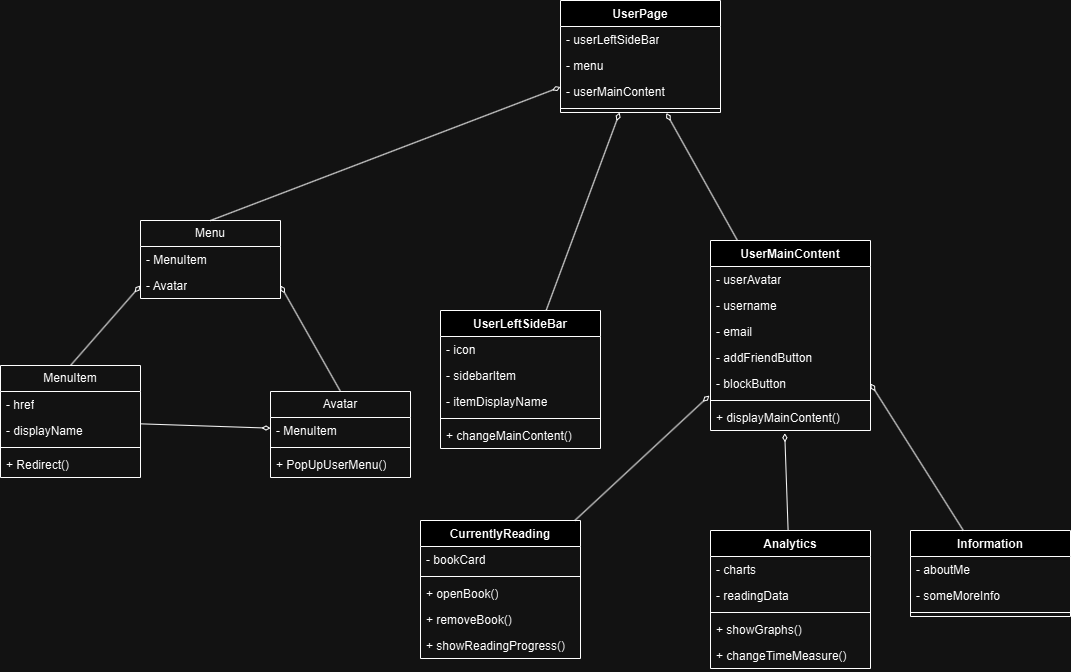
* **Login/Register pages**
  + These are the group of pages with the same target of helping users access the service.
  + The website will require the user to login to access the main contents/services.
  + Users without accounts could switch back and forth between login and registering for an easy account-creating process.
  + All information must be sent to the server using POST requests to prevent any security error or injection attack.



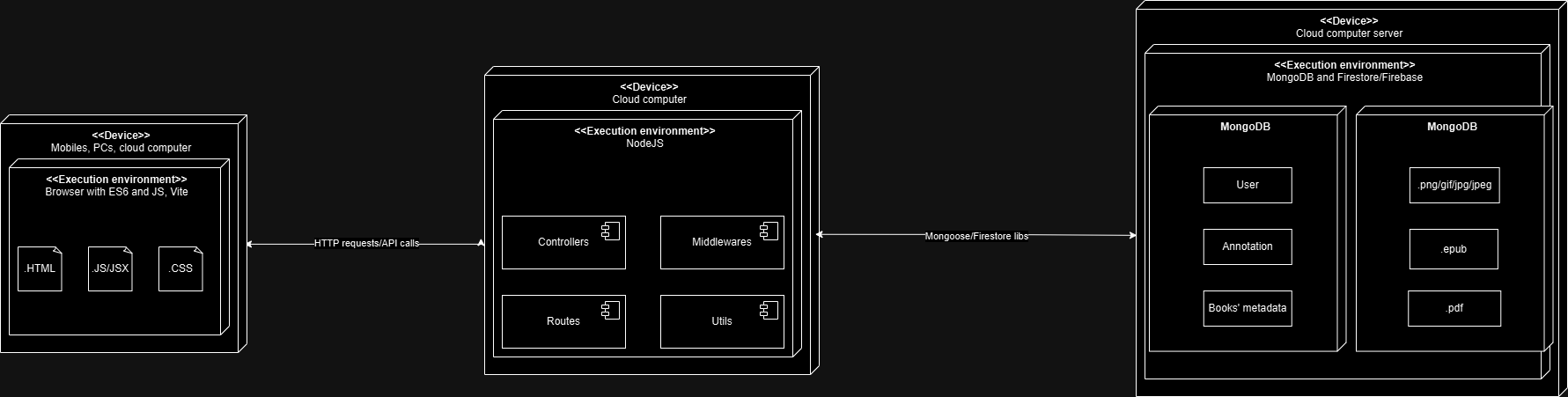
* **Reading page**
  + Reading page shows the viewer the documentation they required, with a menu for customization and accessibility.
  + Reading page also contains a hamburger menu that can be opened or closed, for a better reading experience.
  + The component will use GET requests, because it will create accessible URLs, therefore better user experiences



* **User page**
  + User page is to display the user’s data and analytics.
  + User page reuse the Menu component in the main page for synchronization and unification in designing.
  + The LeftSideBar is used to navigate the tabs, namely CurrentlyReading, Analytics and Information.
  + UserMainContent is to display the content of each tab in the LeftSideBar respectively and accordingly.
  + User page will use the combination of GET and POST requests, based on the need of security and accessibility.

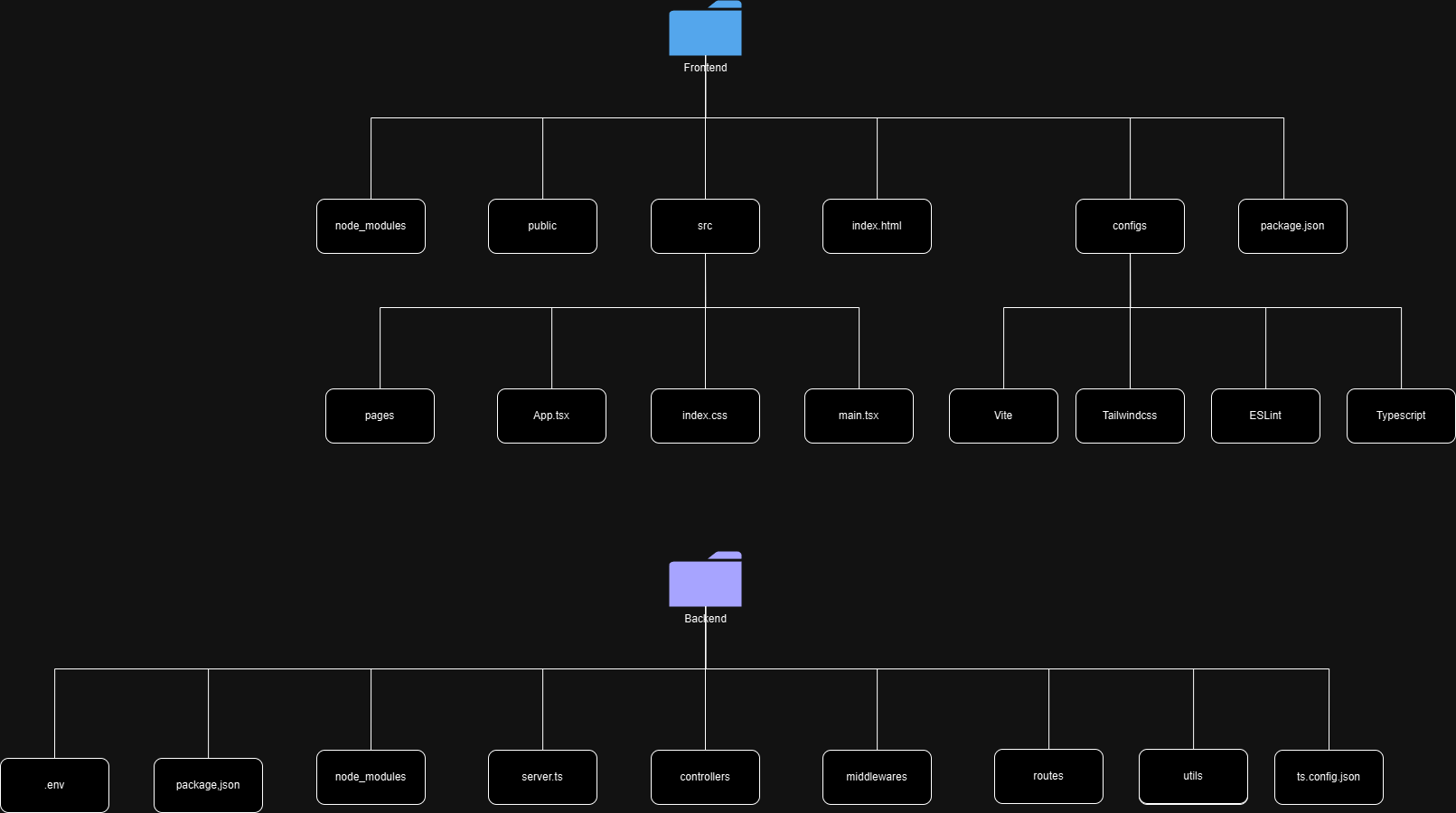


# Deployment



* **Client side:**
  + **Devices**: mobiles, personal computers, etc. In summary, whichever devices that have a browser which support ES6 and Javascript, preferably modern browsers such as: Firefox, Edge or Google Chrome. The frontend will be focusing mostly on the user interface and user experience on personal computers first.
  + **Execution environment:** The client side will run on browsers, compiled by built-in Javascript compilers such as V8 engine. The code and hosting, rendering will be undertaken by Vite.
  + **Available services**: In the prototype phase, the frontend, which consists of Vite as the running and local hosting environment and ReactJS, will be hosted by free services offered by Vercel. In the beta and final phase, the client will be hosted using a virtual private server (VPS), either be provided by AWS EC2 or AWS VPS or Digital Ocean Droplets. The application will be accessible through a browser which supports ES6 and Javascript. This is made possible by using the Vite’s local hosting and the port forwarding feature of a VPS, which likely be the default feature. Another option is Platform as a Service (PaaS), which provides a digital, cloud-computing playground for development, deployment, debugging and managing application’s lifecycle. An example is AWS Elastic Beanstalk
  + **Content:** The final content served to the client will be HTML, CSS and JS files, which are results of packaging, transpiling from TS to JS, bundling, compiling, rendering, etc from Vite.
* **Server side:** 
  + **Device:** User cannot access server side, the application will be hosted on a VPS or a PaaS with no port forwarding and concealed, private IP address for security reasons.
  + **Execution environment**: NodeJS will be the primary environment to run JS applications and services.
  + **Available services:** much like client side, server will also can be hosted by using a VPS such as AWS EC2 or a PaaS such as AWS Beanstalk, although AWS EC2 is much preferred as it offers customizability.
  + **Content:** the content of the backend will consist of JS files, categorized into controllers, middleware, routes, services, utilities.
* **Database:**
  + **Device:** hosted on cloud-provider, VPS or cloud computers.
  + **Execution environment:** database will be stored as JSON inNoSQL environment, and queries will be executed by Object Document Mapper (ODM)
  + **Available services:** NoSQL database provider also supports cloud hosting built-in service, such as Mongo Atlas for MongoDB or Firestore for Firebase. ODMs are provided through libraries such as Mongoose or Prisma.
  + **Content:** Mongodb holds multiple collections of documents, which follow the designing of ERD and the model logical view above. While the Firestore storage is for files and binary objects, which could be categorized into buckets based on usage or attributes.

# Implementation View



* **Frontend**
  + **node\_modules**: the folder is essential for storing third-party modules and libraries, as well as the ReactJS library.
  + **public**: the directory to store public static assets for Vite, which will be shared among components and pages.
  + **src**: the source code of components and pages
    - **pages**: store the TSX code of pages, which are the core of the application.
    - **App.tsx**: the main page of the application, serving, calling all pages and creating routes as well as contexts.
    - **main.tsx:** the core component of ReactJS, to render and apply ReactJS root on DOM to allow ReactJS render React components, as well as applying strict mode.
    - **index.css**: main style sheet which dictates the general style for all components and pages, to create a unified styling.
  + **index.html**: static HTML file, serve as the original DOM for React root to act on.
  + **package.json**: contains project’s details, such as name, author, license, etc, as well as all packages that need to be installed and their version. Also includes, script to run the project.
  + **config files:** configurations for libraries such as Vite, Tailwind, ESLint and Typescript.
* **Backend**
  + **server.ts:** main file to run the ExpressJS server, set up middlewares, configuration for the server. It also listens to API calls, HTTP requests.
  + **controllers:** store controllers for pages, which handle requests, process data, and respond with resources.
  + **middlewares:** folder to store Typescript files which define custom middlewares, which will handle and process data, then pass those data into another middleware or controller, and export them as modules.
  + **routes:** define active routes for the APIs of the application. Routes should map the URLs to the responsible controllers.
  + **utils**: utilities of the application. Helper functions that are exported as modules and will be used in either controllers or middlewares.
  + **services:**
  + **ts.config.json:** config file for Typescript
  + **package.json:** file store the server details, as well as packages for development.
  + **.env:** file to store environment variables. Usually used to store secrets and sensitive information such as email addresses or API keys.
  + **node\_module**s: store libraries and module necessary for development, might be generated by package.json