Eötvös Loránd University

Faculty of Informatics

Dept. of Media & Educational Technology

Personalized Dashboard "Your Personal Space"

Supervisor:
Horváth Győző
Associate professor

Author:
Shepelenko Mykhailo
Computer Science BSc

Budapest, 2025

Contents

Chapter 1

Introduction

Your Personal Space ("**YPS**" for short) is an application designed to enhance the productivity of individuals engaged in intellectual work. The app helps users concentrate on their primary tasks while minimizing distractions. At the same time, **YPS** allows users to stay in control of everything they need outside of work - Chat, Notes, Calendar, News and Weather.

The motivation for creating this project came from a situation I encountered every day. I had to focus on work, while also needing to take notes, stay in touch with colleagues and friends, keep up with the latest news, record sudden events in the calendar, and be ready to leave for a meeting at any moment—knowing what to wear for it. I used many different apps for these tasks, constantly switching between them. This disrupted my focus and significantly decreased my productivity.

I use a monitor and a laptop for work. Usually, the laptop remains open and functions more like a "picture." I believe it is impossible to achieve full concentration while using two screens simultaneously. Secondary but essential applications cannot be comfortably placed on the laptop screen in a way that keeps them always in focus and convenient to use. That's when the idea was born! **Your Personal Space** is a unique solution that addresses the challenge of efficiently utilizing screen space and boosting work productivity.

Intellectual work requires flexibility in choosing the workplace, devices, and their number. My project can be opened in just a couple of clicks! Every device supports a browser, and every standard browser supports my application. The user doesn't need to download anything. They simply need to open the website and log in. Access is available from any device at any time, and work can continue instantly, as all data

is stored on the server.

Thus, **YPS** is an excellent tool for anyone with internet access who needs a high-quality way to manage the vast amount of information we receive in today's world.

Chapter 2

User documentation

Your Personal Space is a Web Application that can be run using modern browser. User can use Chrome, Safari, Firefox, Opera and Microsoft Edge to run it. App can be accesed by link `http://localhost:5173/`.

The application is lightweight and does not require any installation or powerful hardware. It works smoothly on Windows, macOS, and Linux operating systems. It also supports Android and iOS mobile platforms through the browser. No setup is required — user just need to visit the URL and start using the application.

For the best experience, it is recommended to use the latest version of a supported browser. The app is fully responsive and adapts to any screen size, from mobile phones to desktop monitors. Minimum recommended screen resolution is 360x640 pixels.

When user clicks on the link, one redirects to Login Page that is shown below.

2.1 Login/Signup

This section as about Authorization into **YPS** application . There are several ways supported.

2.1.1 Login

Login page is needed to log in to the system or go to the registration page to create a new accoun. User will see dark or light version of **YPS** application depending on default mode preferences set by system or browser:

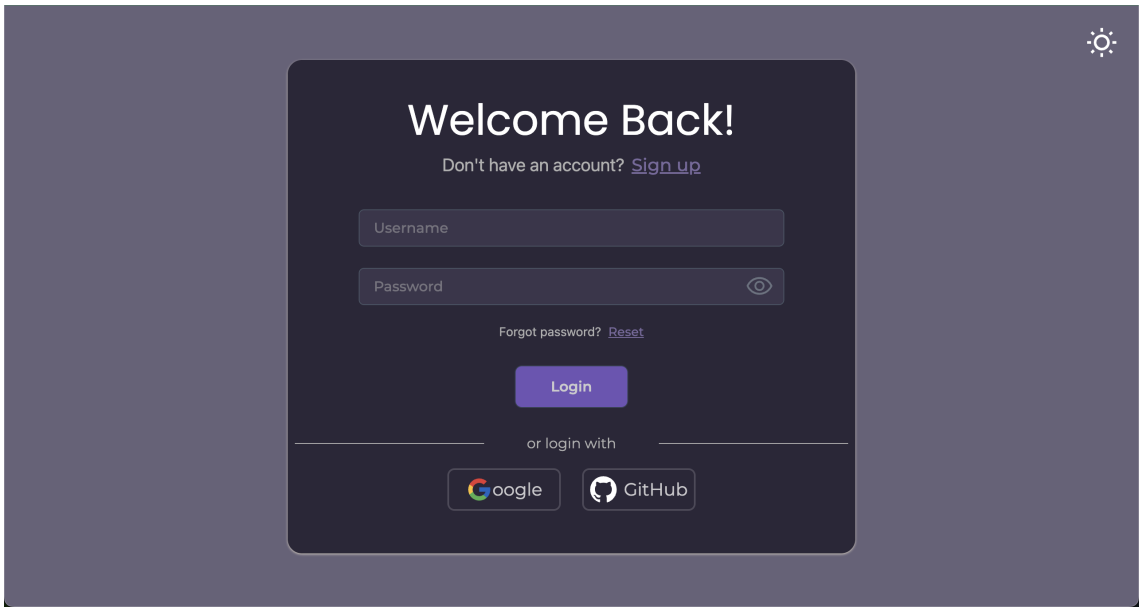


Figure 2.1: Login Dark View

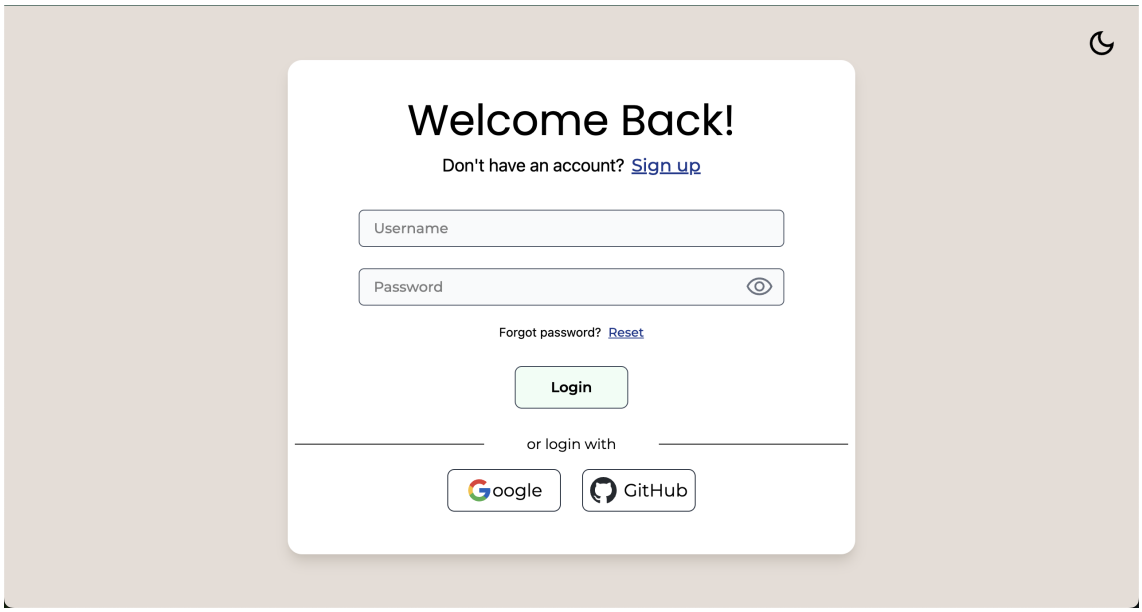
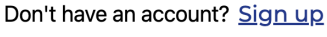


Figure 2.2: Login Light View

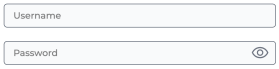
User has several options on Login Page:

- Toggle theme - 

  Change between light and dark theme modes for better visibility

- Sign up - 

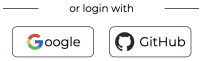
  Create a new account by clicking the link

- Login manually - 

  Enter username and password manually

- Reset Password -

  Reset password if it is lost

- Login using Google or GitHub - 

  Use Google or GitHub account to log in easily

If user decides to create a new account, one will be redirected to *Signup Page*. Otherwise user needs to enter their username and password and click on Login button to be redirected to the Home Page. In case of error, User will get this kind of error:

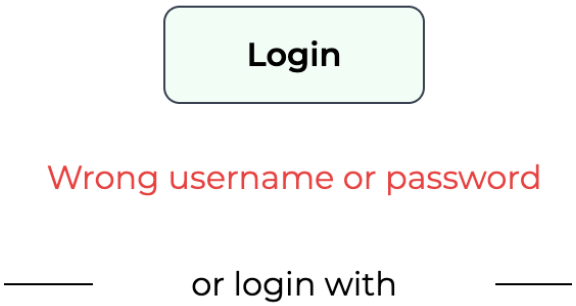


Figure 2.3: Login error

When user clicks on text field to enter username or password, app shows hints to match a format. We are often confused when try to remember needed password variation. **YPS** helps you with that! It looks like that:

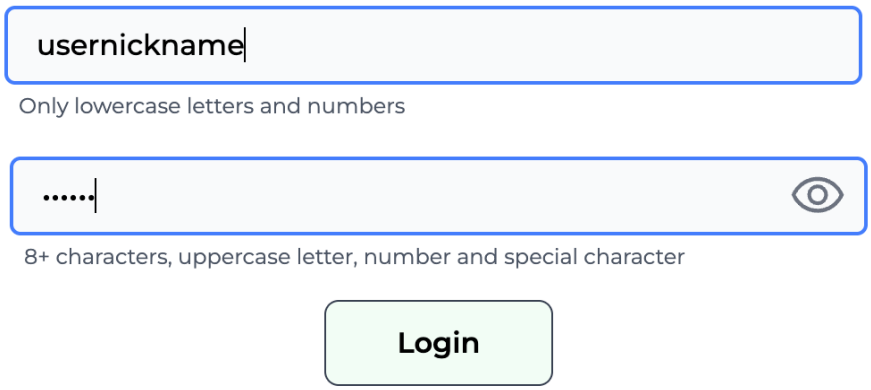


Figure 2.4: Hints

2.1.2 Signup

User can create an account by clicking on the Sign up link located on the Login Page. One will be redirected to the Signup Page where several fields are required to be filled. This page also supports both dark and light modes, adapting to system preferences:

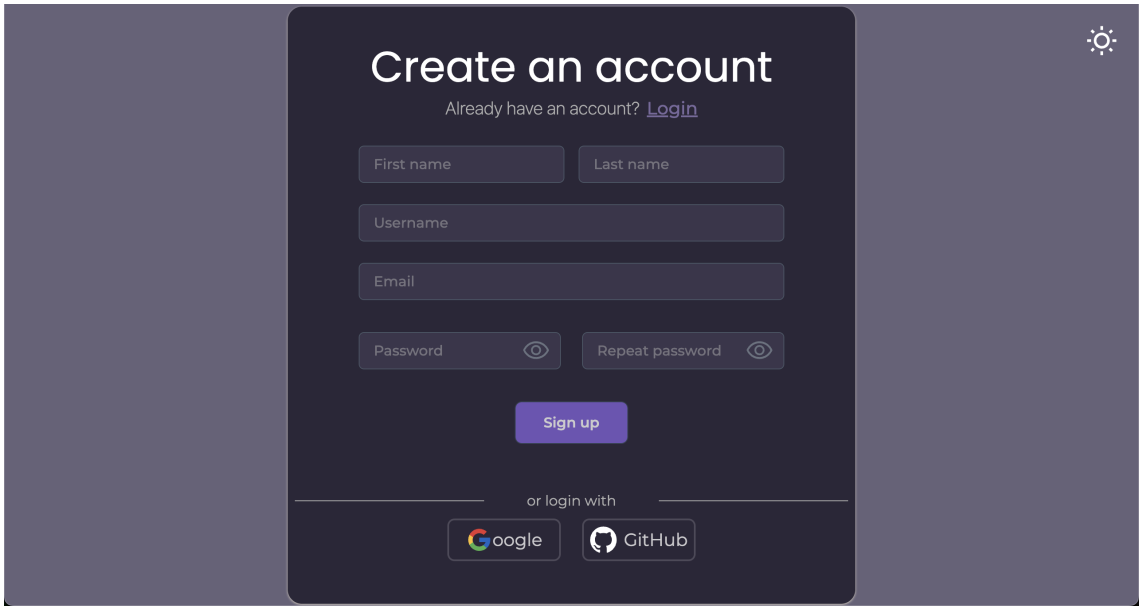


Figure 2.5: Signup Dark View

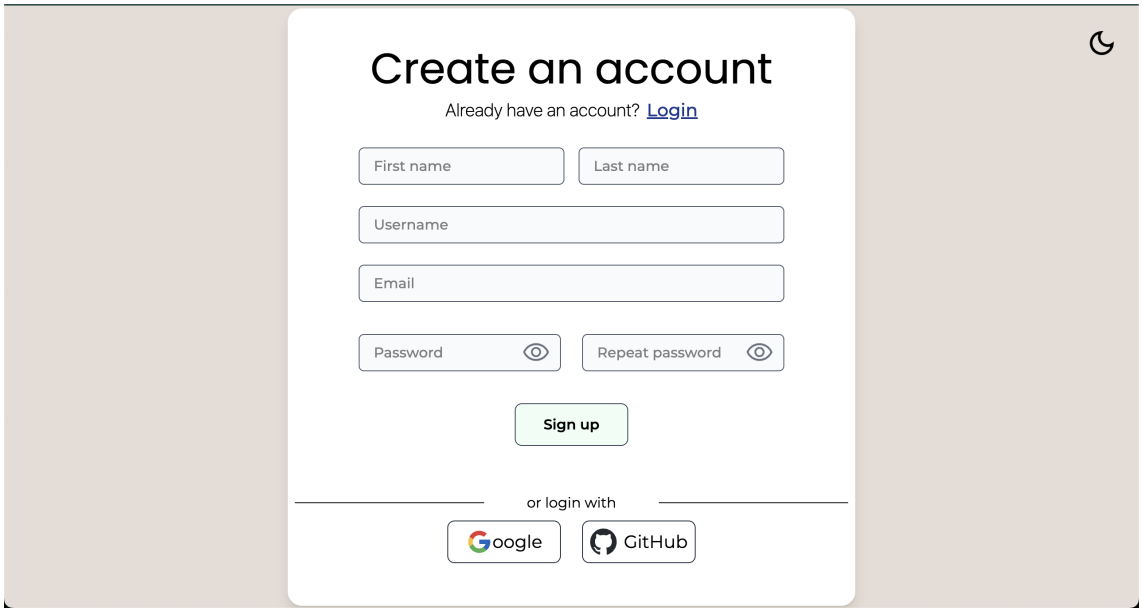


Figure 2.6: Signup Light View

The following inputs must be filled out by user:

- First name and Last name

  Only Latin letters are allowed, between 3 to 20 characters each. Numbers and special characters are not accepted.
- Username

  Must contain only lowercase letters and numbers, with a minimum of 5 characters.
- Email

  Should be in standard email format (e.g., example@domain.com). Invalid format will show an error.
- Password and Repeat password

  Password must be between 8 to 20 characters, and include at least one uppercase letter, one lowercase letter, one digit, and one special character. Both password fields must match.

If user enters incorrect values in any field, a helper text will appear under that field describing the issue. Password hints are also shown to guide the user in creating a strong password. It looks like:

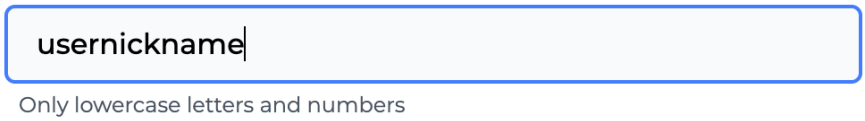


Figure 2.7: Signup hints

After entering all required information, user clicks the **Sign up** button. After that user is sent to email verification page. Code is already sent to email and user has to verify it here.

If an error occurs during signup (not matching passwords or invalid format of any field), a friendly error message is shown as seen below:

If user is already exists in database or passwords are not the same, the following errors will be shown:

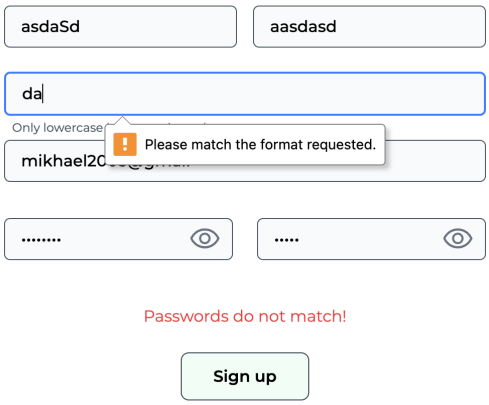


Figure 2.8: Signup error example

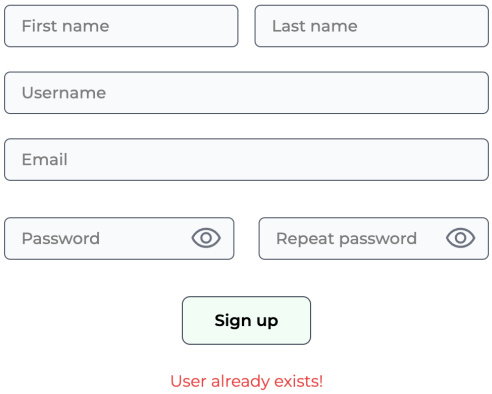


Figure 2.9: Signup error example

2.1.3 Email confirmation

User is required to check email that was entered on signup page and enter it in "Code from email" field and press "Confirm" button.

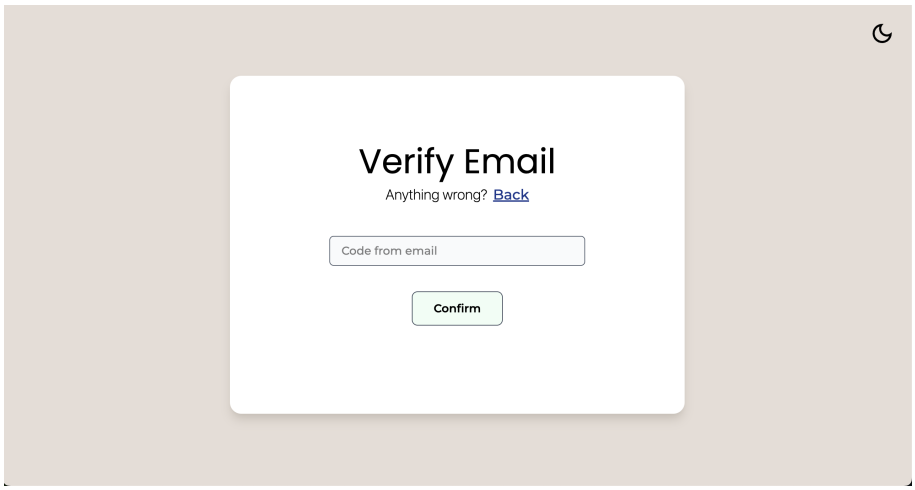


Figure 2.10: Email verification

If the code is correct, user will be sent to the home page.

If the code is wrong, user will see error message.

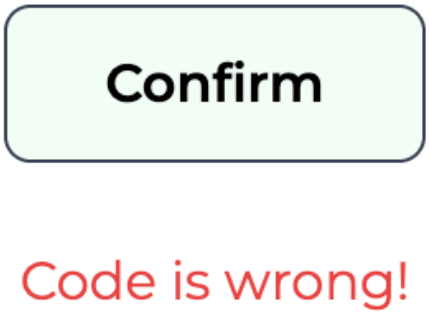


Figure 2.11: Code error example

If all validations pass, user account is created and system redirects to the Home Page. Otherwise, user will be prompted to fix the highlighted issues.

Signup also supports authentication via Google or GitHub for convenience. These options appear below the form.

Figure 2.12: Google/GitHub login options

After successful signup, user can immediately begin using the application with full access to all features.

Finally user will be sent to a Home Page that contains all available widgets at the same time on the screen and Navigation Bar.

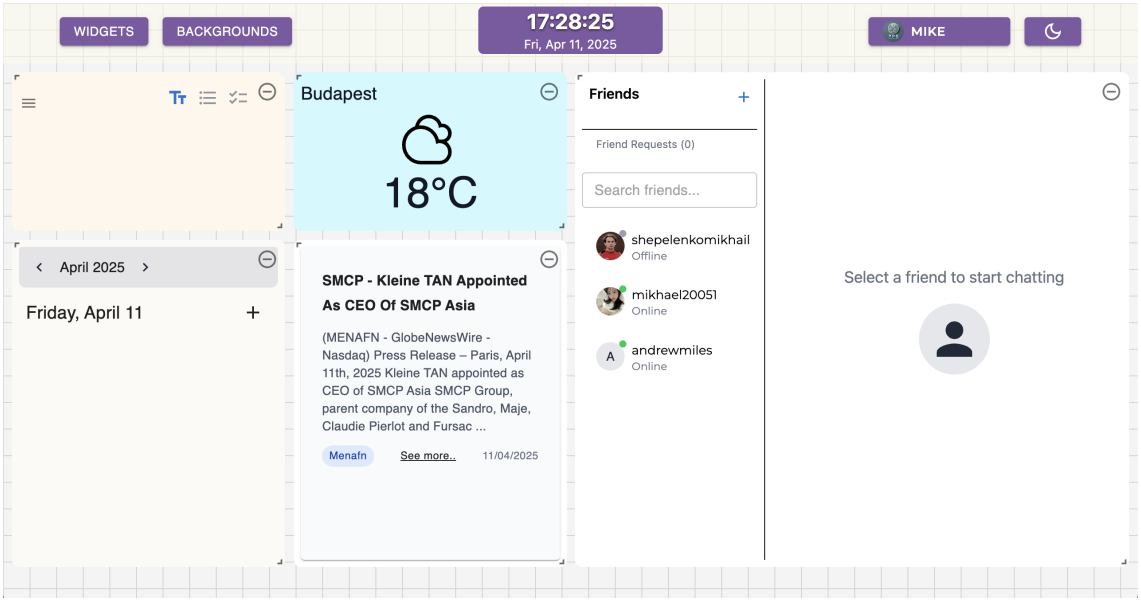


Figure 2.13: Application Light View

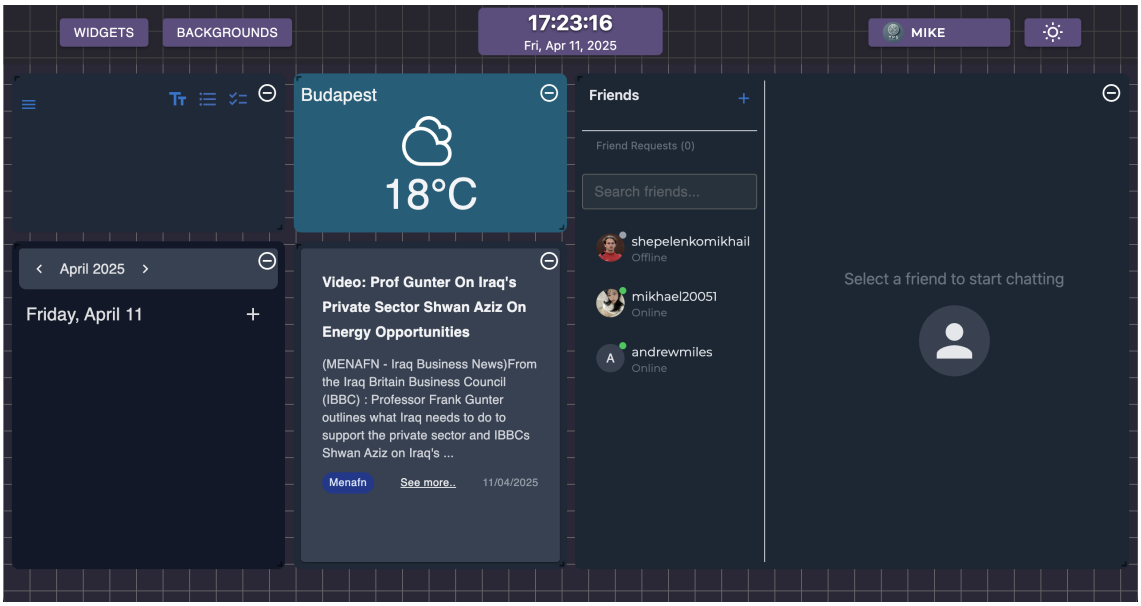


Figure 2.14: Application Dark View

2.2 Navigation and customization

After successful authentication user can see the Home Page that consists of Header and Dashboard. User can customize experience using elements from the Header that includes 4 buttons and Clock.

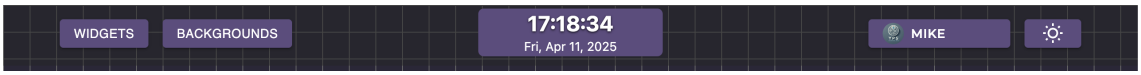


Figure 2.15: Header - Desktop version

Figure 2.16: Header - Mobile version

Mobile version Header consists of Burger icon, Clock and Theme toggle button. Some buttons are hiden inside menu that can be opened by pressing on Burger Icon. Fuctionality of all buttons is the same as in Desktop version.

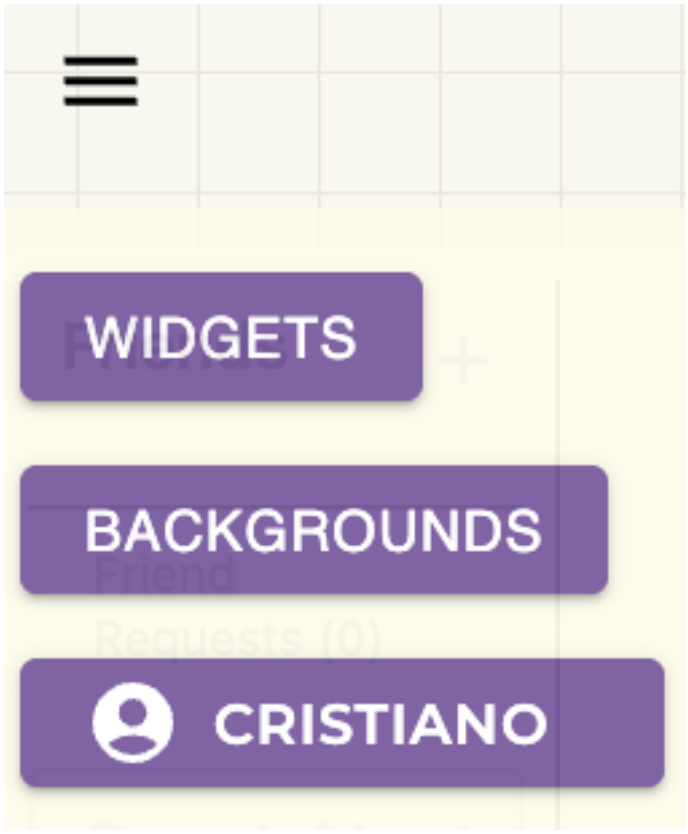


Figure 2.17: Opened menu - Mobile version

2.2.1 Widgets

First button is "Widgets" button that opens menu where all available widgets are listed.

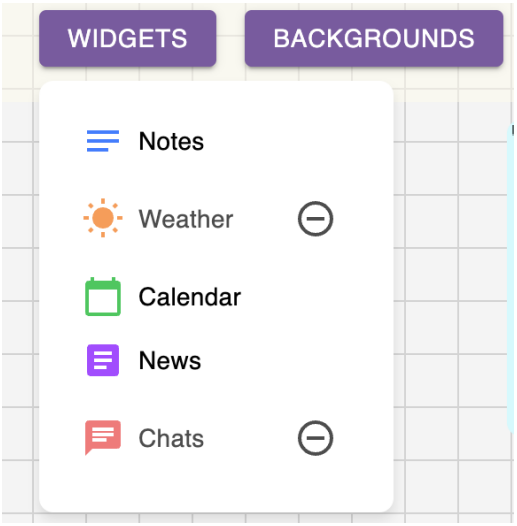


Figure 2.18: Widgets button and menu - Light version

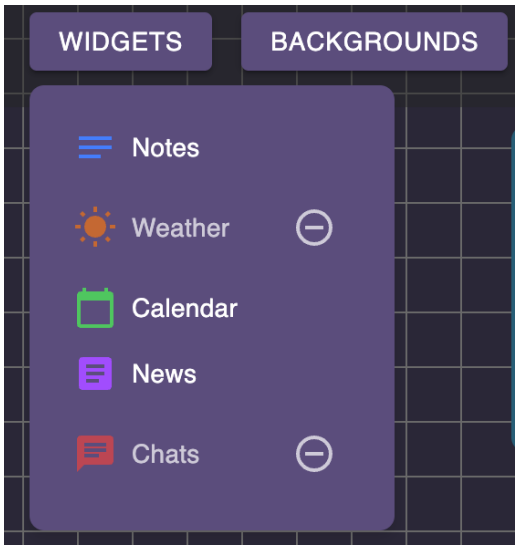


Figure 2.19: Widgets button and menu - Dark version

Click on wiget's name or icon closes the menu and adds it to Dashboard. Remove button that looks like "No entry" sign is appeared only next to widgets that are already opened. Click on it will keep menu opened and remove chosen widget.

2.2.2 Background

When the user clicks on the Backgrounds button in the header, a menu appears that allows uploading a custom background image. The interface supports both light and dark themes. In both versions, the menu includes an Upload Image button with a cloud icon and a drop zone for file selection.

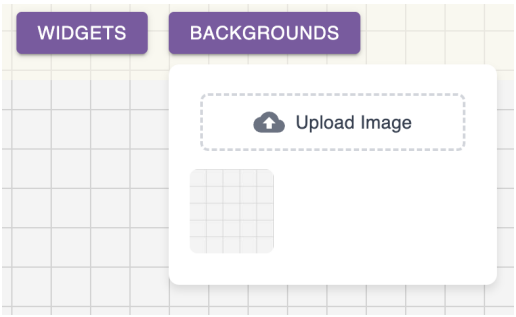


Figure 2.20: Backgrounds button and menu - Light version

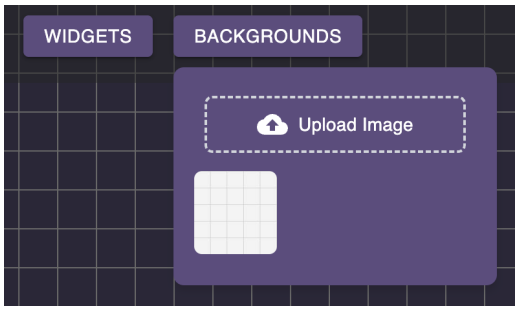


Figure 2.21: Backgrounds button and menu - Dark version

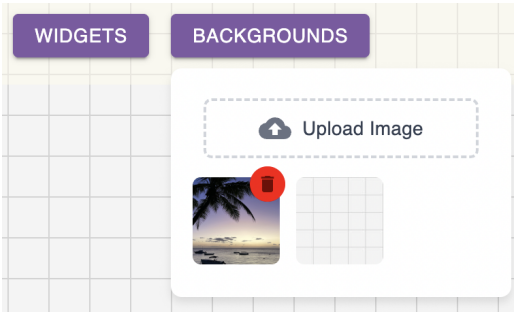


Figure 2.22: Backgrounds button and menu containing image

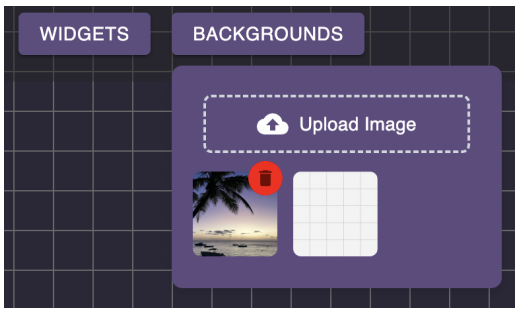


Figure 2.23: Backgrounds button and menu containing image

User can drag and drop an image or click the button to select a file manually. After uploading, the image appears in the preview grid. This feature allows further personalization of the workspace by setting a visual background. User can remove an image from the list using "Trash" icon.

2.2.3 Clock

The clock component dynamically displays the current time and date and adjusts its layout depending on the device. In the desktop version, the clock shows the full time including hours, minutes, and seconds, along with the complete date in the format "Day, Month Date, Year". In the mobile version, to preserve space and maintain clarity on smaller screens, the clock simplifies its display by showing only the hour and minute.

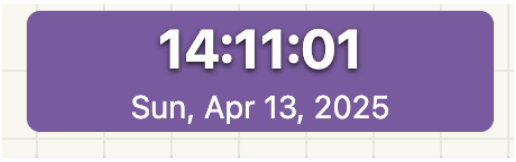


Figure 2.24: Clock for desktop version

Figure 2.25: Clock for mobile version

This responsive design ensures that the time display remains accessible and visually optimized across different devices.

2.2.4 Profile

The profile section provides access to the user settings menu, which includes options for account configuration and logging out. The design adapts to the active

theme (light or dark). In both versions, clicking on the profile button—labeled with the username—reveals a dropdown menu. This menu includes:

- Settings - *Navigates to the user's account configuration page.*
- Logout - *Safely ends the current user session and redirects the user to the login page.*

The profile menu is visually distinct in each theme. The light version features a white dropdown against a light background, while the dark version uses deeper shades for better visibility and consistency.

2.2.5 User settings

The User Settings section allows users to view and update their personal account information. The layout and visual styling of this section automatically adjusts to the active theme (light or dark), maintaining visual consistency across the interface.

In both theme versions, the user sees their profile avatar, username, email, first and last names, and password fields. By default, all fields are disabled to prevent accidental changes. To edit any of the fields, the user must press the pen icon located next to each input. This action enables the corresponding field for editing.

The form enforces the same validation rules as during the registration process, ensuring consistency and security of user data.

The avatar icon is also interactive. Clicking on the avatar allows the user to upload a new profile picture.

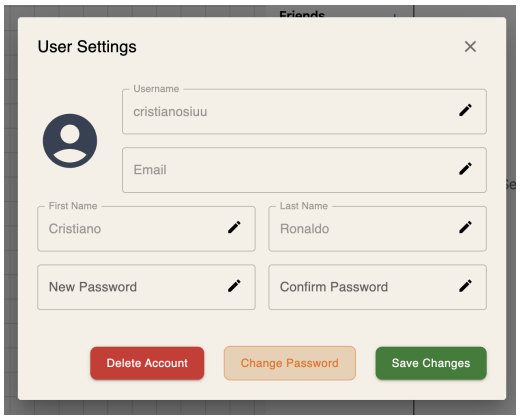


Figure 2.26: User settings - Light version

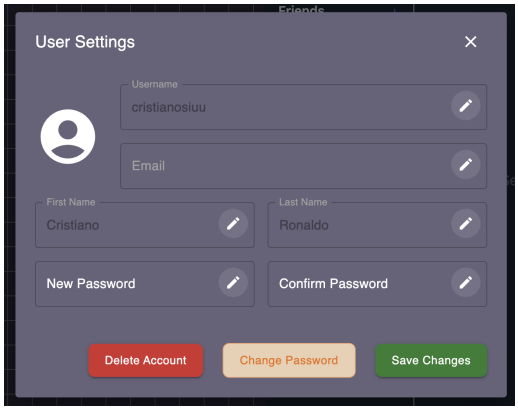


Figure 2.27: User settings - Dark version

Changes are only saved after pressing the **Save Changes** button. This ensures that users can review their modifications before confirming any updates.**Change**

Password button will change the password if both fields for it are filled. **Delete account** button asks for confirmation and deletes account if user accept it.

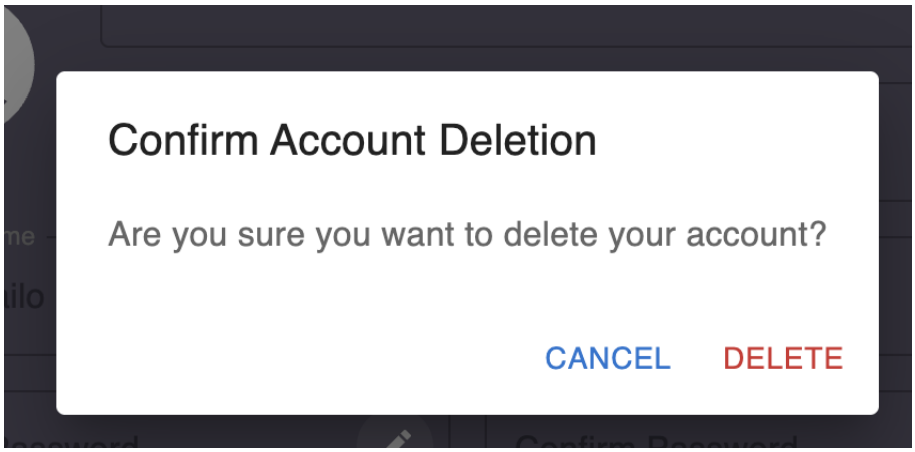


Figure 2.28: Account delete confirmation

2.2.6 Theme toggle

The theme toggle button allows users to switch between light and dark modes to improve visual comfort and personalization.

- Light version - *Displays a crescent moon icon, indicating a switch to dark mode.*
- Dark version - *Displays a sun icon, indicating a switch to light mode.*

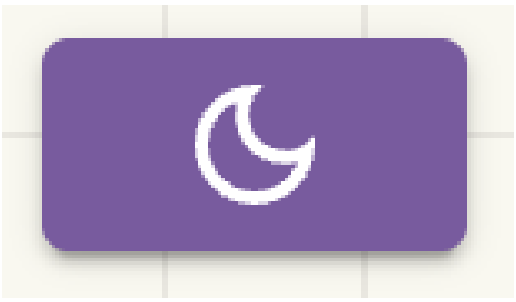


Figure 2.29: Theme toggle button - Light version

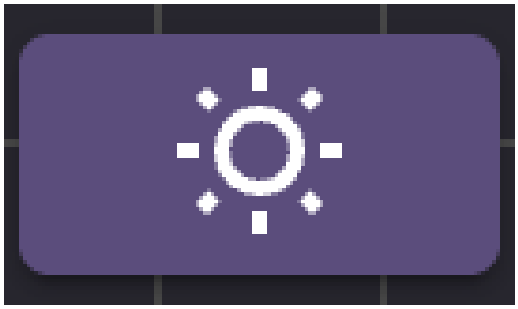


Figure 2.30: Theme toggle button - Dark version

This toggle is consistently styled with a purple background and adjusts its icon based on the current theme.

2.3 Widgets Dashboard

The Notes Widget allows users to create, edit, and organize text-based notes directly on the dashboard. It supports two display modes (light/dark) and adapts to

screen size with a compact layout for mobile devices. Each note has an icon on the right top corner that removes it from Dashboard.

2.3.1 Notes Widget

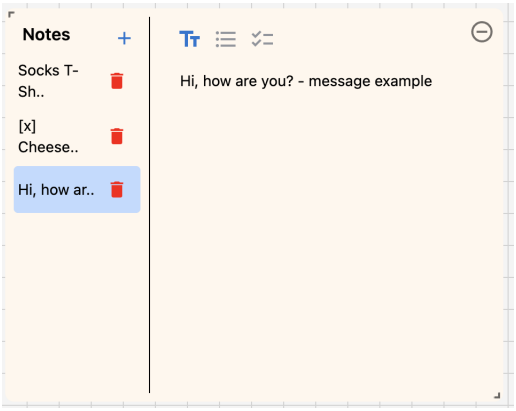


Figure 2.31: Notes widget - light mode

Figure 2.32: Notes widget - dark mode

There are three supported modes: Simple Note, Ordered List, and Checklist. The user can switch between them by clicking the icons located in the top-right corner. Note titles are set automatically to reduce the user's cognitive load. The left-side menu displays a list of all notes. Each note has a "Trash" icon that allows the user to delete it. Clicking on a note opens it on the right side.

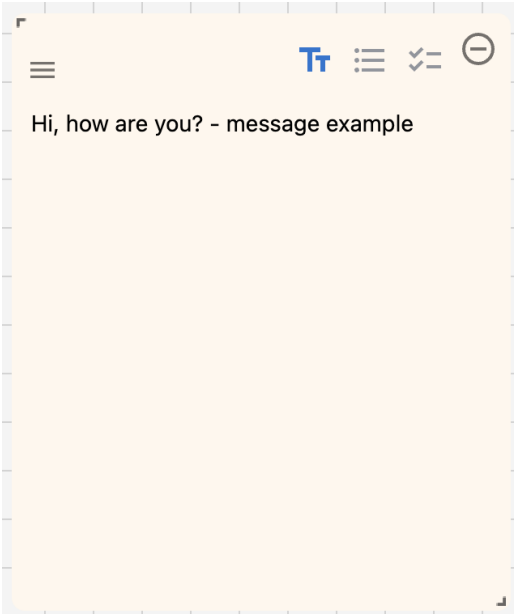


Figure 2.33: Compact view of notes page

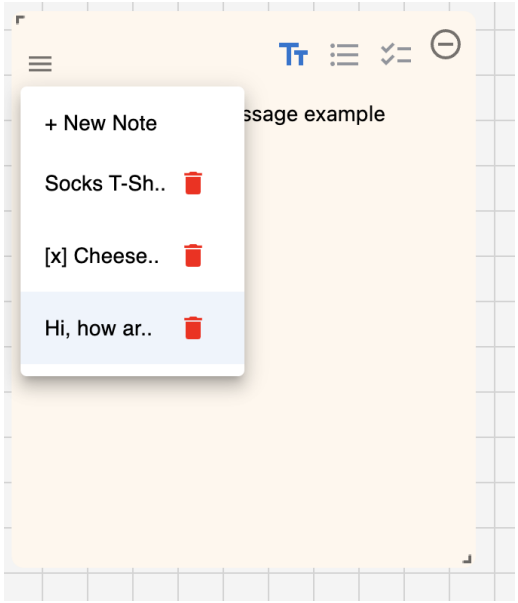


Figure 2.34: Compact view with menu open

The small version of the widget hides the list of notes, which can be accessed by pressing the burger icon in the top-left corner.

2.3.2 Calendar Widget

The Calendar Widget enables users to view and manage their schedules in multiple formats: monthly, weekly, and daily views. It automatically adjusts to the selected theme (light or dark) for a consistent visual experience across the dashboard. Clicking on a cell in month view will open a menu for adding an event fot the day of this cell. Month view can be accessible only when widget has big height enough to show whole month. Otherwise only week and day view are accessible.

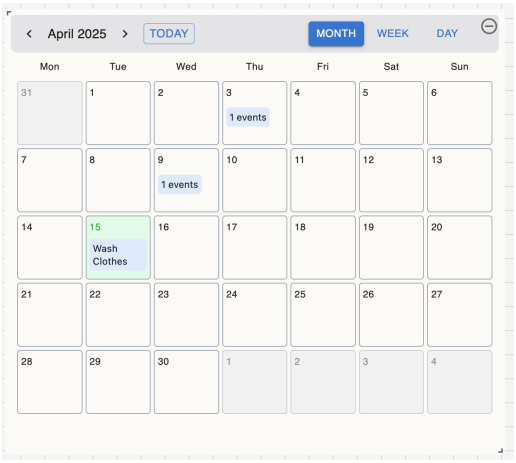


Figure 2.35: Calendar - Light version

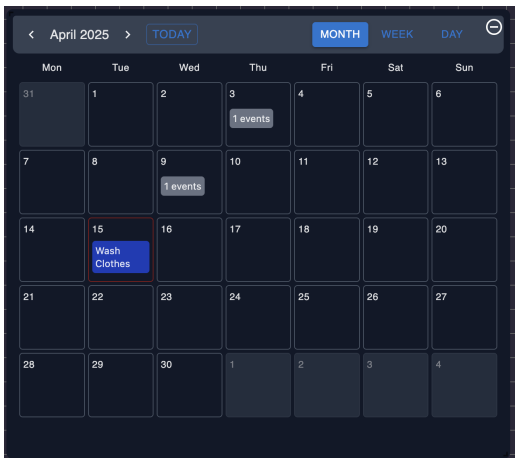


Figure 2.36: Calendar - Dark version

In the day view, events are presented in a scrollable timeline with hourly slots for easy scheduling. The layout is optimized for readability and allows precise time-blocking. Clicking on "Edit" icon or anywhere on the event section opens Edit menu includind all data about this event. "Delete" icon deletes a selected event. A dedicated Add/Edit Event menu allows users to quickly enter event details such as title, description, and time. Events can be updated or removed using intuitive icons.

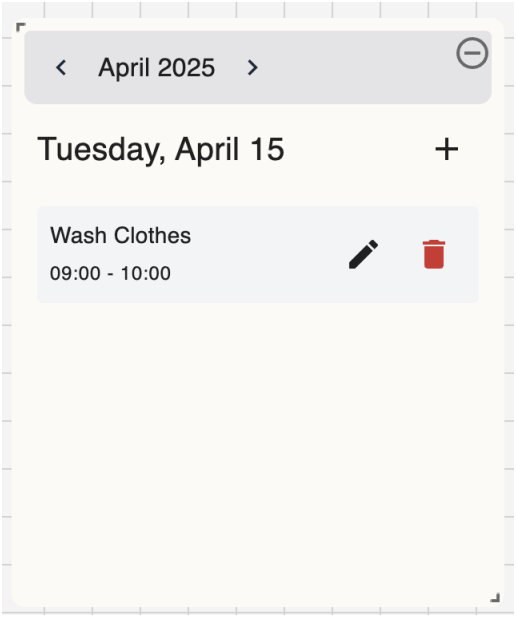


Figure 2.37: Calendar - day version

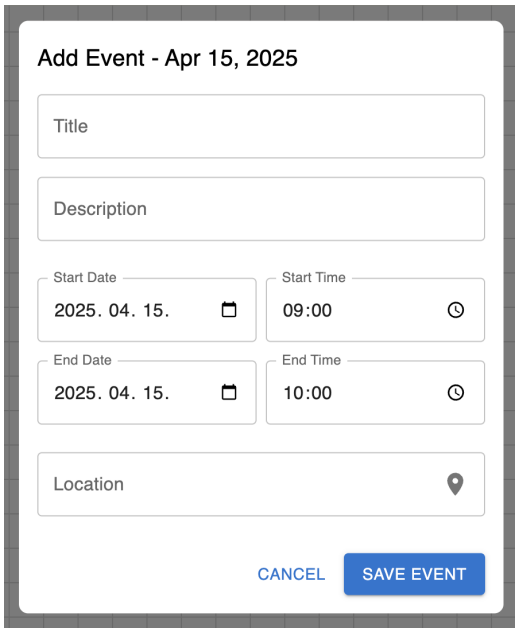


Figure 2.38: Add/Edit event menu

When calendar is wide enough for showing a whole week, "Week" button becomes availabele. Clicking on this button changes a view mode and shows a whole week. User can click on event to change it or on any point on the day box to add a new event (Add event menu will be opened). "Today" button is also shown only on wide version of the widget.

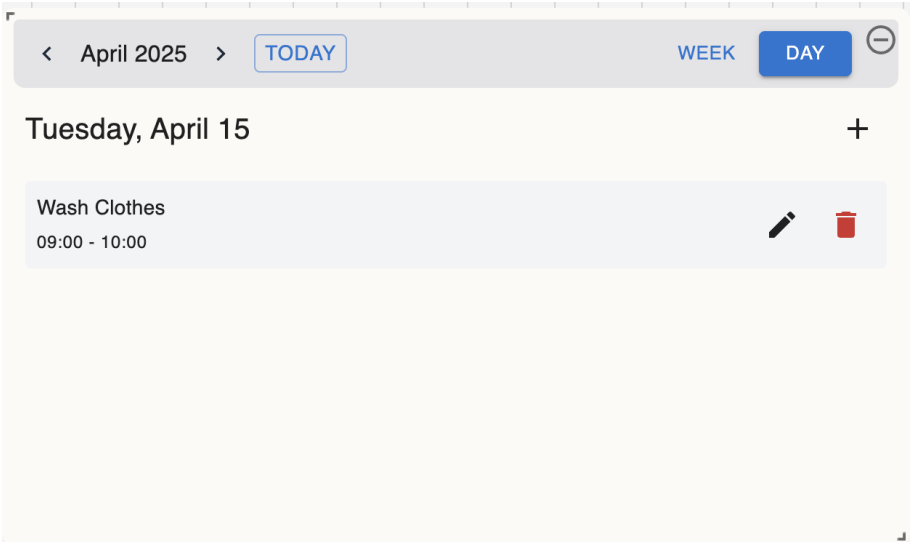


Figure 2.39: Calendar - wide day version

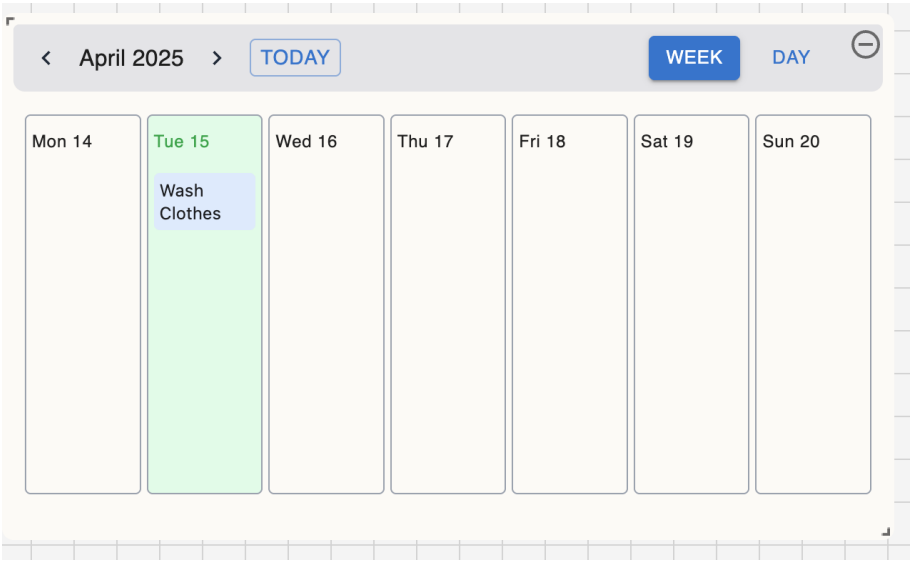


Figure 2.40: Calendar - week version

The week view provides an overview of all scheduled activities across the week, facilitating better planning. The wide layout adjusts to larger screens, displaying multiple columns with high clarity.

2.3.3 Weather Widget

The Weather Widget provides real-time temperature and weather condition updates for the user's current location. Upon initialization, the system attempts to detect the user's location automatically if user allows it in browser. In case of an error or inability to determine the location, the default city is set to London. The widget displays temperature in Celsius along with relevant weather icons. When the widget has sufficient height, it also presents additional information such as the "Feels like" temperature and a five-hour forecast.

Different sizes of the widget are available to accommodate various layouts. The largest version offers the most detailed view, including hourly forecasts, while smaller versions prioritize essential data such as the current temperature and weather icon. Users can change their city by clicking the pencil icon located next to the city name, which opens a dialog box for manual input.

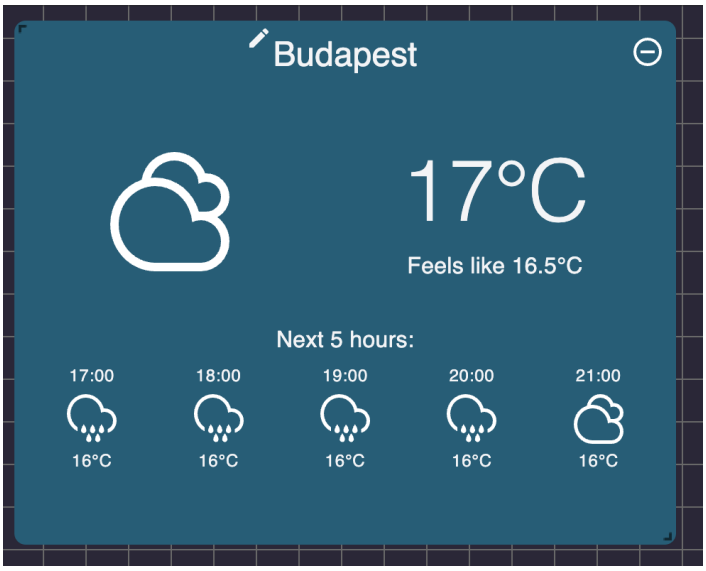


Figure 2.41: Weather big version

Figure 2.42: Weather small version

The widget adapts responsively to screen size and layout constraints. In its extra small version, only essential details—city name, temperature, and weather icon—are shown, ensuring clarity on compact displays. The “edit” icon remains present across all versions, offering consistent access to the city change feature. Clicking on "Edit" icon will open the menu for city changing. Upon activation, the user is prompted to enter a new city name, with confirmation and cancellation options available.

Figure 2.43: Weather extra small version

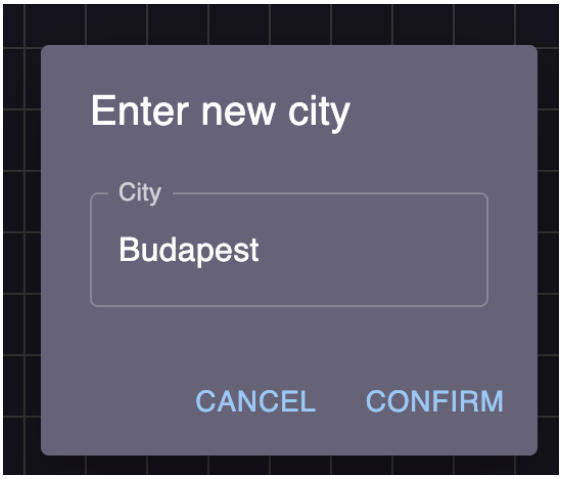


Figure 2.44: City change menu

This widget offers a compact and intuitive experience while maintaining functional consistency across its various sizes. Automatic location detection, fallback defaults, and simple user input mechanisms ensure accessibility and personalization for all users.

2.3.4 News Widget

The News Widget provides users with a brief overview of the latest headlines tailored to their location. This component is available in both light and dark themes, adjusting automatically based on the active theme setting.

News content is sourced and refreshed every five minutes to ensure users are presented with the most up-to-date information. By default, news articles are displayed in English and are relevant to the user's current location, which is determined automatically. If the system is unable to detect the user's location, the widget defaults to showing news relevant to London. Users can manually change their location via the Weather Widget, which then updates the news accordingly.

Each news entry displays a headline, a short summary, the source, and the publication date. A **See more** button is available for each article, redirecting users to the full story on the original site.

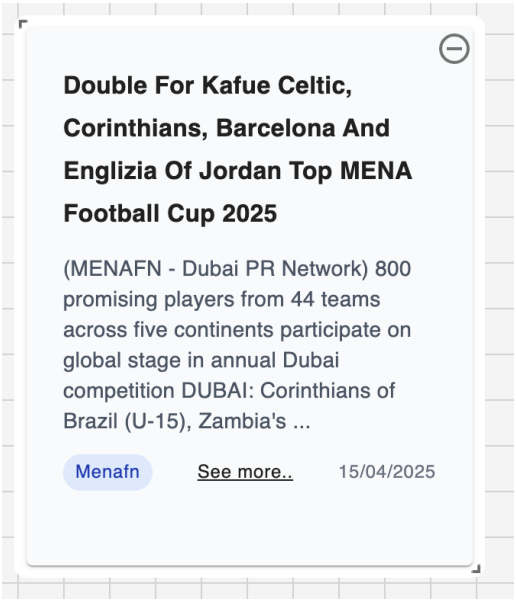


Figure 2.45: News - light version

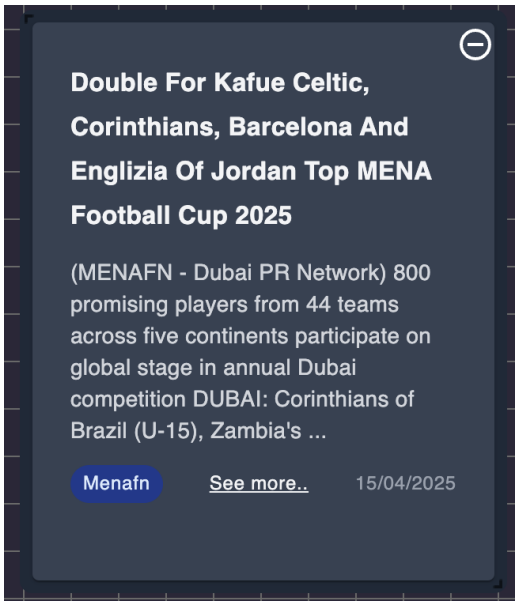


Figure 2.46: News - dark version

The layout ensures readability in both themes, with high contrast for text and clear visibility of interactive elements.

2.3.5 Chats Widget

The Chats Widget allows users to communicate in real-time and manage their friend connections directly from the application interface. It supports both light and dark themes for better user experience across different lighting conditions.

In the main layout, the left panel displays the friends list along with any pending friend requests. The right panel is used for active conversations, which become visible upon selecting a friend. Figures 2.47 and 2.48 demonstrate the appearance of the widget in light and dark modes respectively.

Users can attach several files in the same message up tp 60mb in total and up to 15mb each. Videos, pictures and audio content has a preview interface in chat and link to download it. All users has presence indicator under their nicknames and color indicator on the right top corner of their avatars.

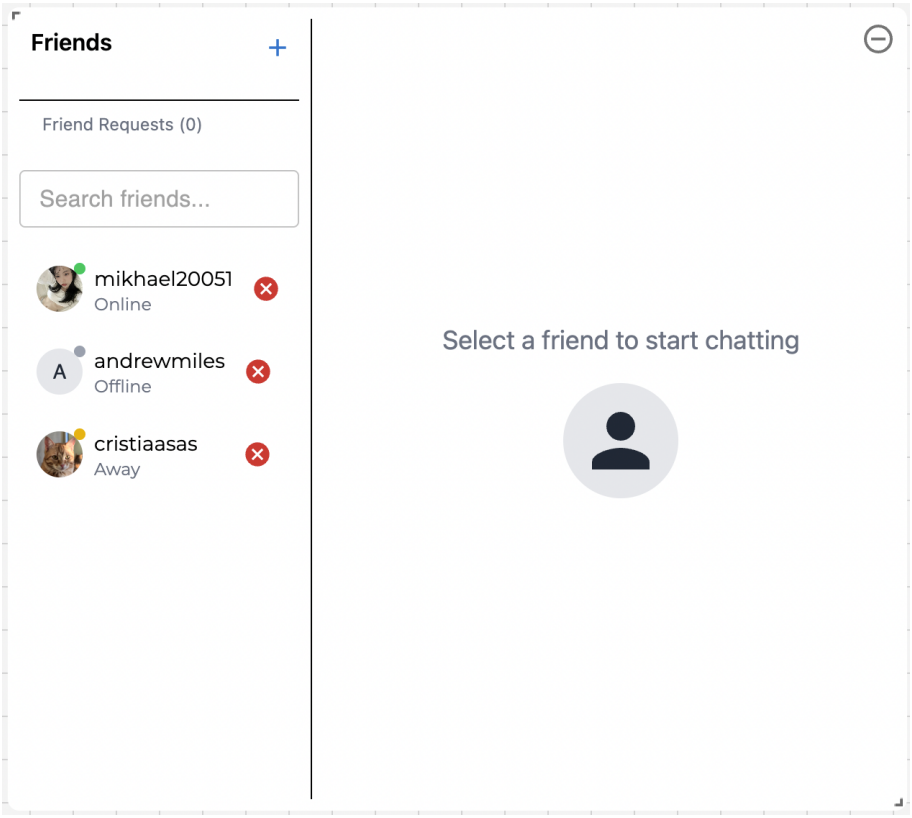


Figure 2.47: Chats - light version

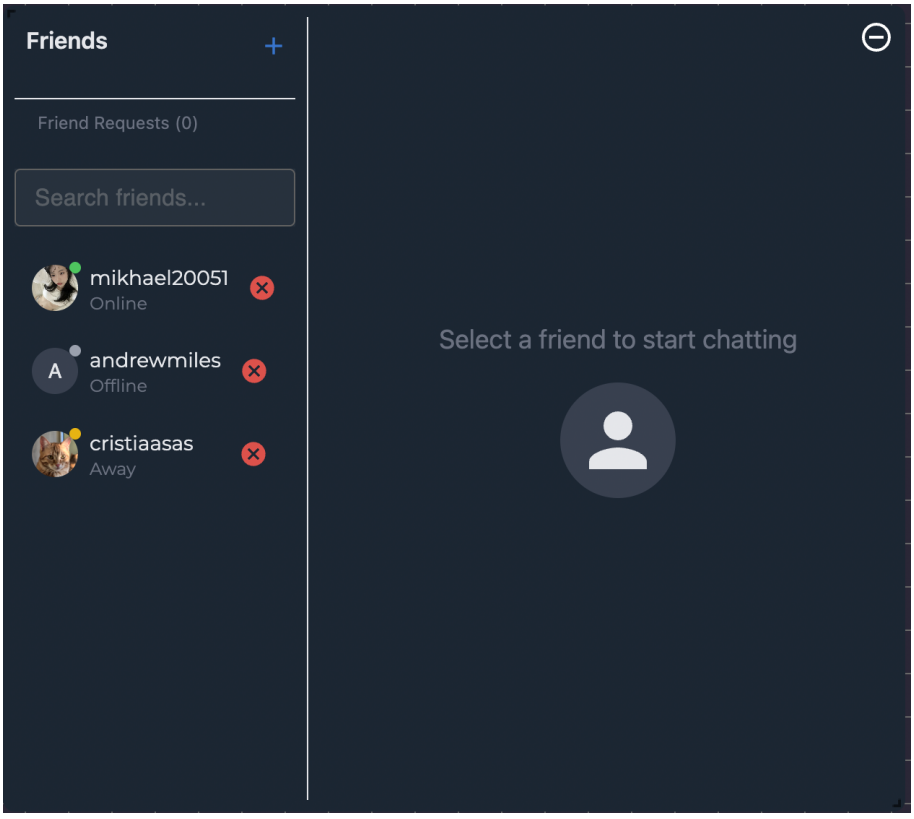


Figure 2.48: Chats - dark version

Users can send and manage friend requests within the same widget. Incoming

requests are listed at the top of the friends section. Each request shows the sender's name and the date it was sent, with accept and reject options. Figure 2.49 shows an example of a received friend request.

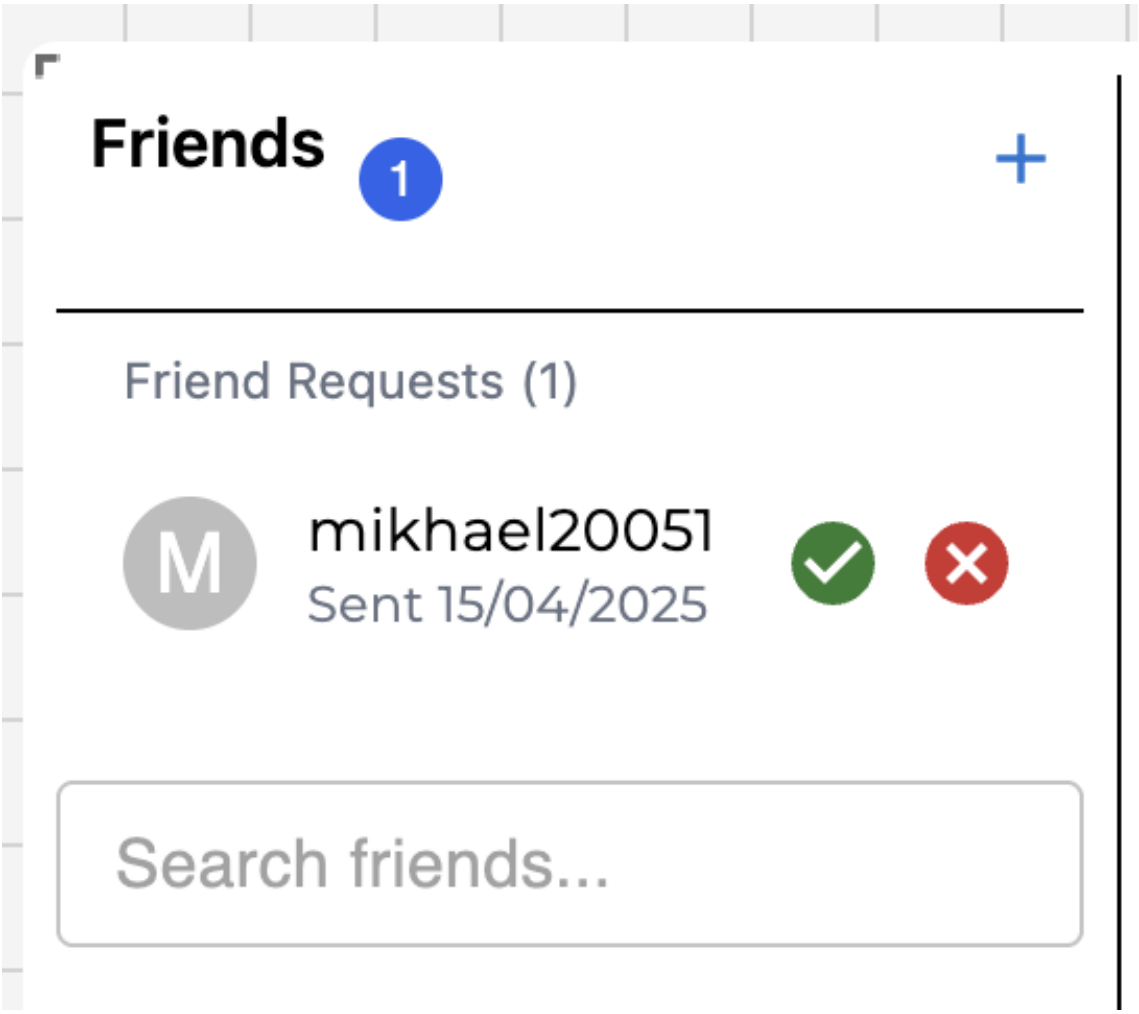


Figure 2.49: Friend request example

To find new users, the friend search bar can be used. As users type a name or part of it, matching profiles are shown with an option to send a friend request. Figure 2.50 illustrates this functionality.

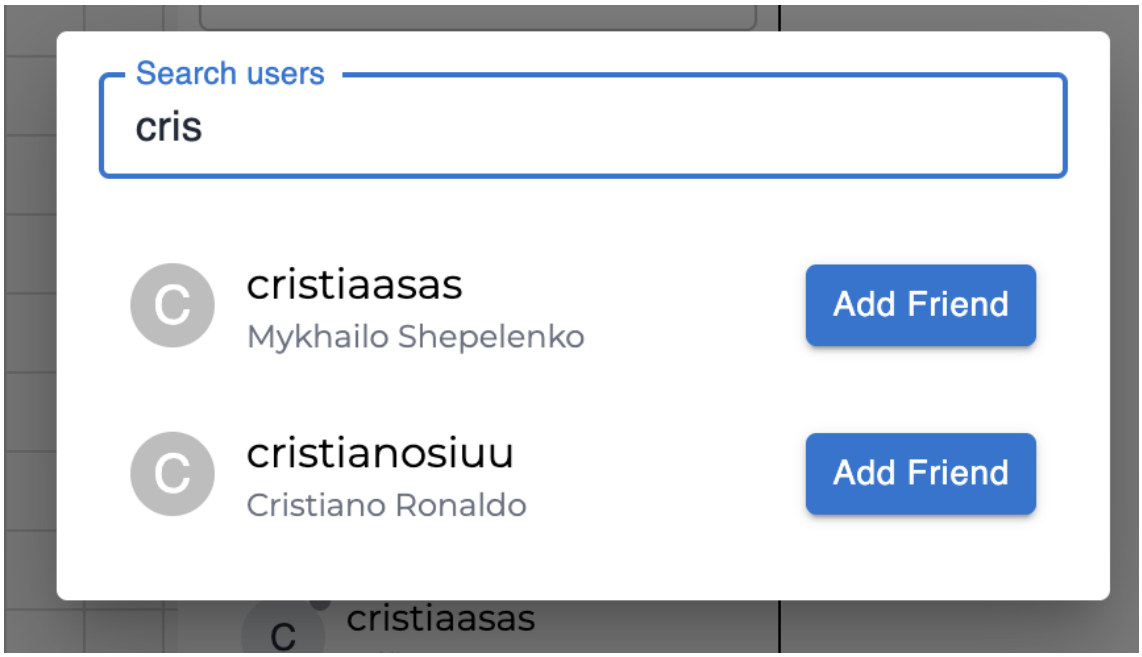


Figure 2.50: Friends search example

On smaller screen sizes or compact layouts, a dropdown menu is used to access the friends list and friend request section. This view also includes an option to add new friends. Figure 2.51 shows this smaller chat version in action.

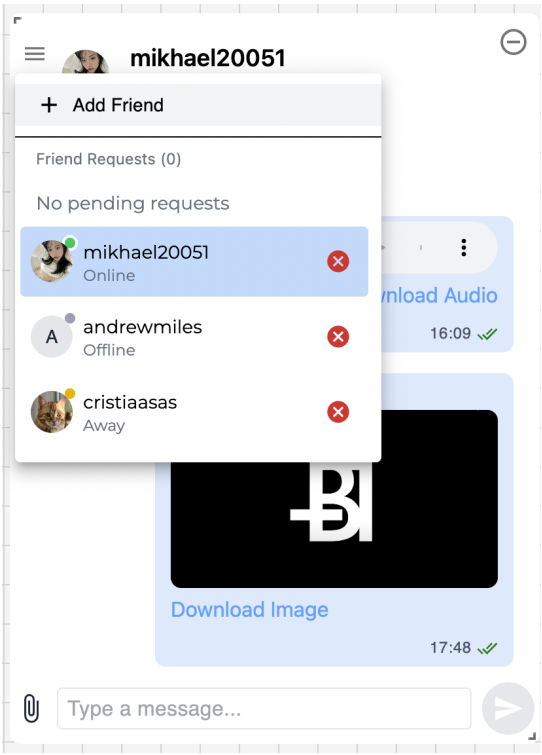


Figure 2.51: Small chat version with opened friends list menu

When a friend request is sent, the system provides immediate feedback via a

notification banner. Similarly, if a sent request is rejected or accepted, the user is notified accordingly. These interactions are shown in Figures 2.52 and 2.53.

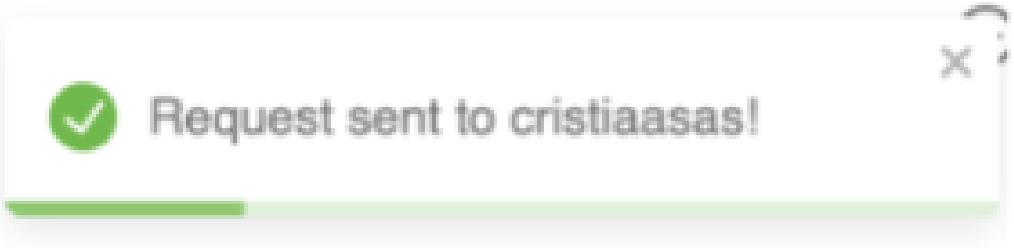


Figure 2.52: Request sent notification

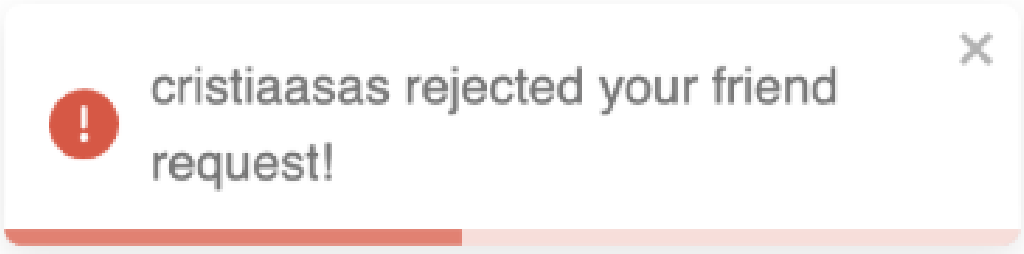


Figure 2.53: Rejected request notification

All communication and friend interactions happen in real-time, ensuring that users receive updates and responses instantly for a smooth and efficient messaging experience.

Friends can be removed from the list permanently using red Remove button.

Figure 2.54: Red button to delete friend

Green notification will be shown in case of successfull delete and Blue notificaiton will be shown to the user who was deleted from someone's friend list.

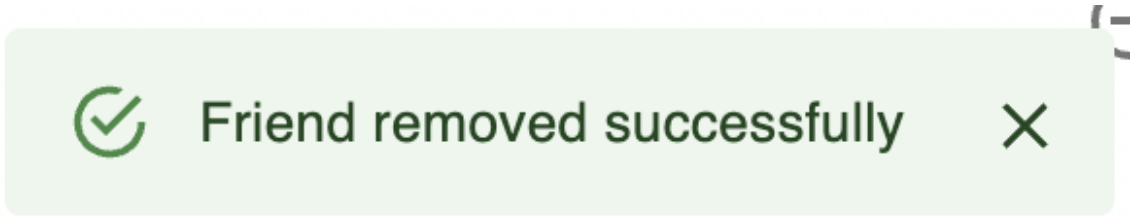


Figure 2.55: Friend delete notification

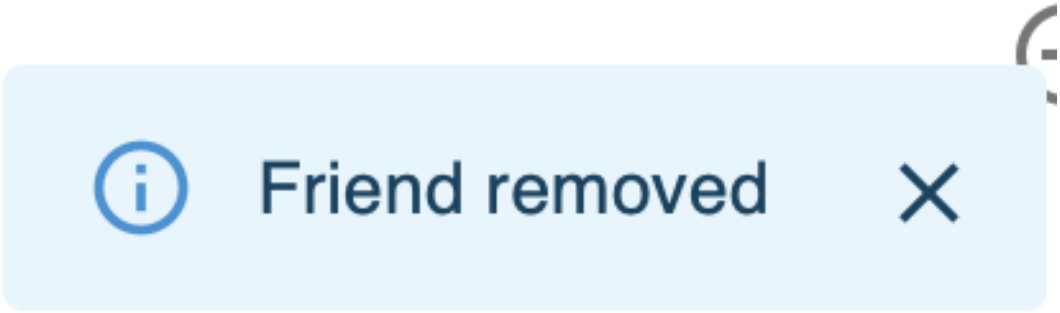


Figure 2.56: User was deleted from someone's friend list

2.3.6 Mobile version

User can use only one widget at a time on mobile versions to prevent accidental widgets movement. Chosing a widget from list will replace the current one on the screen and will keep menu open. In case of clicking on wrong widget, user can quickly change it to another one.

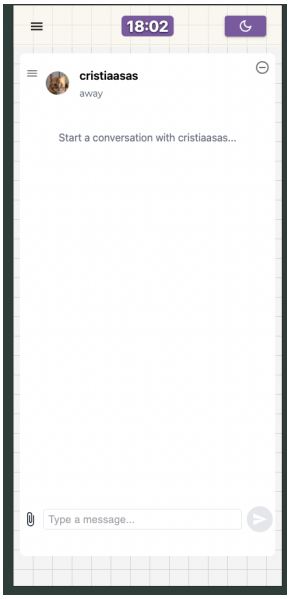


Figure 2.57: Mobile version view

Chapter 3

Developer documentation

3.1 Requirements

This chapter outlines the core requirements of the application from a developer's perspective. It serves as a guide for understanding the expected behavior of the system, both from the user interface and backend sides. The goal is to provide a comprehensive overview of the functional and non-functional elements necessary for developing, maintaining, and extending the application. This includes a detailed list of planned and implemented features, user stories to reflect real-world interactions, and key system qualities like usability, performance, and security. Together, these requirements form the foundation for technical decisions and future development.

3.1.1 Functional requirements - List of functions

Frontend

- Authentication and User Management
  - User login and registration
  - Password reset functionality
  - OAuth integration (Google, GitHub)
  - User profile management that icludes possibility to change avatar, name and surname, username and password.
  - Session management: user's session is managed using JsonWebTocken and user join different socket rooms depending on actions.

  - Proper error handling on each step.

- Dashboard and Widget System
  - Widget layout management: widgets are sored in local storage and updadated on every change to maintain best user's experience across all devices.
  - Widget resizing and repositioning
  - Widget state persistence: widgets can be added using list of widgets and removed use the button on widget or on it's name in the list.
  - Background customization: any image can be set as background, but always can be changed to default one.
  - Dark/Light mode switching keeps best user experience. It takes preferences from browser by default and stored in local storage.

- Weather Widget
  - Current weather display: degrees in Celcius and city
  - Feels like: this section is shown only on the big enough versions of widget
  - Weather forecast: next 5 hours forecast section is shown only on the big enough versions of widget.
  - Location-based weather:
  - Weather condition icons: xorresponding icons are always show current weather state.

- News Widget
  - News article display: articles are always in English and depend on user's location. It us updated every 5 minutes.
  - Article categorization
  - News source management
  - News last update date

- Calendar Widget

  - Event creation and management: calls a menu where user can write all needed details of event and save it (name, description, location, date and time)
  - Jumping to today: button is shown only on day and week versions of the widget.
  - Event editing and deletion
  - Calendar view: day, week and month.

- Notes Widget
  - Note creation and editing
  - Note deletion
  - Note modes: user can have text, ordered list and checked list
  - Automatic note naming: a note's name is beginning of it's content. Naming is automatic to reduce user's cognitive load.

- Chat System
  - Real-time messaging: includes media and audio preview and download option.
  - File sharing: total size of files in one message is 60mb and maximal size of each file is 15mb. Only safe file types can be sent (those that can't break backend by hackers)
  - Message status tracking
  - Typing indicators: read and sent.
  - Friend management: real-time friend requests, nice visualization of friend's list showing avatar, name, status and amount of unread messages. Add and remove friends.
  - Online status tracking

- Geolocation Services
  - Location detection
  - Address to coordinates conversion
  - Coordinates to address conversion

Backend

- Authentication and Authorization
  - User authentication using JWT tokens and session management
  - OAuth integration with Google and GitHub
  - Password encryption using bcrypt
  - Session handling with express-session and secure cookie configuration
  - Role-based access control for different user actions
- Database Management
  - MongoDB integration with Mongoose for data modeling
  - User data persistence with profile information
  - Message storage with real-time updates
  - Event management with calendar integration
  - Note storage with automatic naming
  - Image handling with secure storage
- Real-time Communication
  - WebSocket implementation using Socket.IO
  - Message broadcasting to specific rooms
  - Status updates for online/offline users
  - Typing indicators with real-time updates
  - Friend request, add and remove notifications
  - Chat room management
- External API Integration
  - Weather data fetching from OpenWeatherMap API
  - News aggregation from NewsAPI
  - Geolocation services with address/coordinates conversion
  - Email services for password reset and notifications

  - API key management and rotation

- File Management
  - Image upload and storage with size limits (15MB per file, 60MB total)
  - File type validation for security
  - File serving with proper MIME types
  - File deletion with cleanup
  - Background image management
  - Chat media handling

- Security
  - Input validation using middleware
  - CORS management with proper headers
  - Rate limiting for API endpoints
  - API key management and rotation
  - Secure file upload handling
  - XSS and CSRF protection

- Data Processing
  - Data transformation for API responses
  - Error handling with proper status codes
  - Response formatting with consistent structure
  - Data validation using middleware
  - Logging system for debugging
  - Performance optimization

3.1.2 Functional requirements - User Stories

The following functional requirements describe the main capabilities of the application, organized by user roles using a user-story format.

Here's a comprehensive list of user stories categorized by widgets and features, based on the provided user documentation:

Login/Signup

- As a guest, I want to register with email, Google, or GitHub so I can create a new account.
- As a guest, I want to log in using my credentials or third-party authentication.
- As a guest, I want to reset my password if I forget it.
- As a guest, I want to verify my email address during registration.
- As a guest, I want to see validation hints for username/password requirements.
- As a guest, I want to toggle between light/dark theme on the login page.

Navigation & Customization

- As a user, I want to toggle between light and dark themes.
- As a user, I want to upload custom background images.
- As a user, I want to view the current time and date in the header.
- As a user, I want to access my profile settings.
- As a user, I want to see the widget list, add and remove them.
- As a user, I want to logout.
- As a mobile user, I want to access all navigation options through a compact menu.

User Settings

- As a user, I want to edit my profile information (name, username, email).
- As a user, I want to change my password securely.
- As a user, I want to upload a profile picture.
- As a user, I want to delete my account permanently.
- As a user, I want to see validation errors when editing my profile.

Notes Widget

- As a user, I want to create, edit and delete notes.
- As a user, I want to organize notes in different formats (simple, checklist, ordered list).
- As a user, I want to view all my notes in a sidebar.
- As a mobile user, I want to access my notes through a compact menu.
- As a user, I want my notes to be automatically saved and named.

Calendar Widget

- As a user, I want to view my schedule in day, week, or month view.
- As a user, I want to add, edit and delete events.
- As a user, I want to see event details when I click on them.
- As a user, I want to quickly jump to today's date.
- As a mobile user, I want to see a simplified calendar view.

Weather Widget

- As a user, I want to see current weather for my location.
- As a user, I want to change my location manually.
- As a user, I want to see hourly forecasts when space allows.
- As a user, I want to see "feels like" temperature.
- As a user, I want the widget to adapt to different screen sizes.

News Widget

- As a user, I want to see news headlines relevant to my location.
- As a user, I want to read news summaries in the widget.
- As a user, I want to click "See more" to view full articles.

- As a user, I want news to automatically update every 5 minutes.
- As a user, I want news to adapt when I change my location in the weather widget.
- As a user, I want to move widget only holding areas without text.

Chats Widget

- As a user, I want to send and receive real-time messages.
- As a user, I want to manage my friend list by adding and removing friends.
- As a user, I want to send and receive real-time messages.
- As a user, I want to see message read indicators.
- As a user, I want to send and respond to friend requests.
- As a user, I want to search for new friends.
- As a user, I want to attach files to my messages.
- As a user, I want to see media preview in chat.
- As a user, I want to see presence indicators and unread messages indicators for my friends.
- As a mobile user, I want to access chats through a compact interface.
- As a user, I want to receive notifications for updates in friends and requests lists.

General

- As a user, I want all widgets to adapt to my chosen theme (light/dark).
- As a user, I want to add/remove widgets from my dashboard.
- As a mobile user, I want the interface to adapt to my screen size.
- As a user, I want all my data to persist between sessions.
- As a user, I want to receive visual feedback for my actions.

- As a user, I want to save and restore widgets layout, mode and active widgets.
- As a user, I want to move widgets holding only the parts of widget that do not handle any other actions.

3.1.3 Non-functional requirements

This section outlines the non-functional requirements that define the quality attributes of the system. Unlike functional requirements, which describe specific behaviors or functions, non-functional requirements focus on how the system performs under various conditions.

Performance Requirements

- Response Time
  - The application should respond to user actions within 2 seconds for 95% of requests
  - Real-time chat messages should be delivered within 500ms
  - Widget updates should be reflected within 1 second
  - Weather and news data should be refreshed within 3 seconds

Reliability Requirements

- Availability
  - The system should maintain 99.9% uptime
  - Scheduled maintenance windows should be communicated in advance
  - The system should automatically recover from minor failures
- Data Integrity
  - All user data should be backed up daily
  - No data loss should occur during system updates

Security Requirements

- Authentication
  - All user sessions should expire after 12 hours of inactivity
  - Password and username requirements should enforce minimum complexity
- Data Protection
  - All sensitive data should be encrypted at rest
  - File uploads should be scanned for malware

Usability Requirements

- User Interface
  - The interface should be responsive and work on devices from 320px to 1920px width
  - All interactive elements should have clear visual feedback
  - The system should maintain consistency across all widgets
  - Color contrast should meet WCAG 2.1 AA standards
  - All errors must be shown as feedback
- Accessibility
  - All images should have appropriate alt text
  - The system should support screen readers

Maintainability Requirements

- Code Quality
  - Follow DRY principle and write understandable code
  - Documentation should be updated with each major change

Compatibility Requirements

- Browser Support
  - The application should work on the all well-known popular browsers
  - The application should work on the at least few latest versions of the all well-known popular browsers
- Device Support
  - The application should be fully functional on mobile devices
  - Touch interactions should be optimized for mobile users
  - Widget layouts should adapt to different screen sizes

Localization Requirements

- Weather data should use appropriate units based on region
- News content should be regionally relevant
- Time zones should be handled correctly across all features

3.1.4 Use-case Diagram

The following use-case diagram visually represents the main user roles and their interactions with the system:

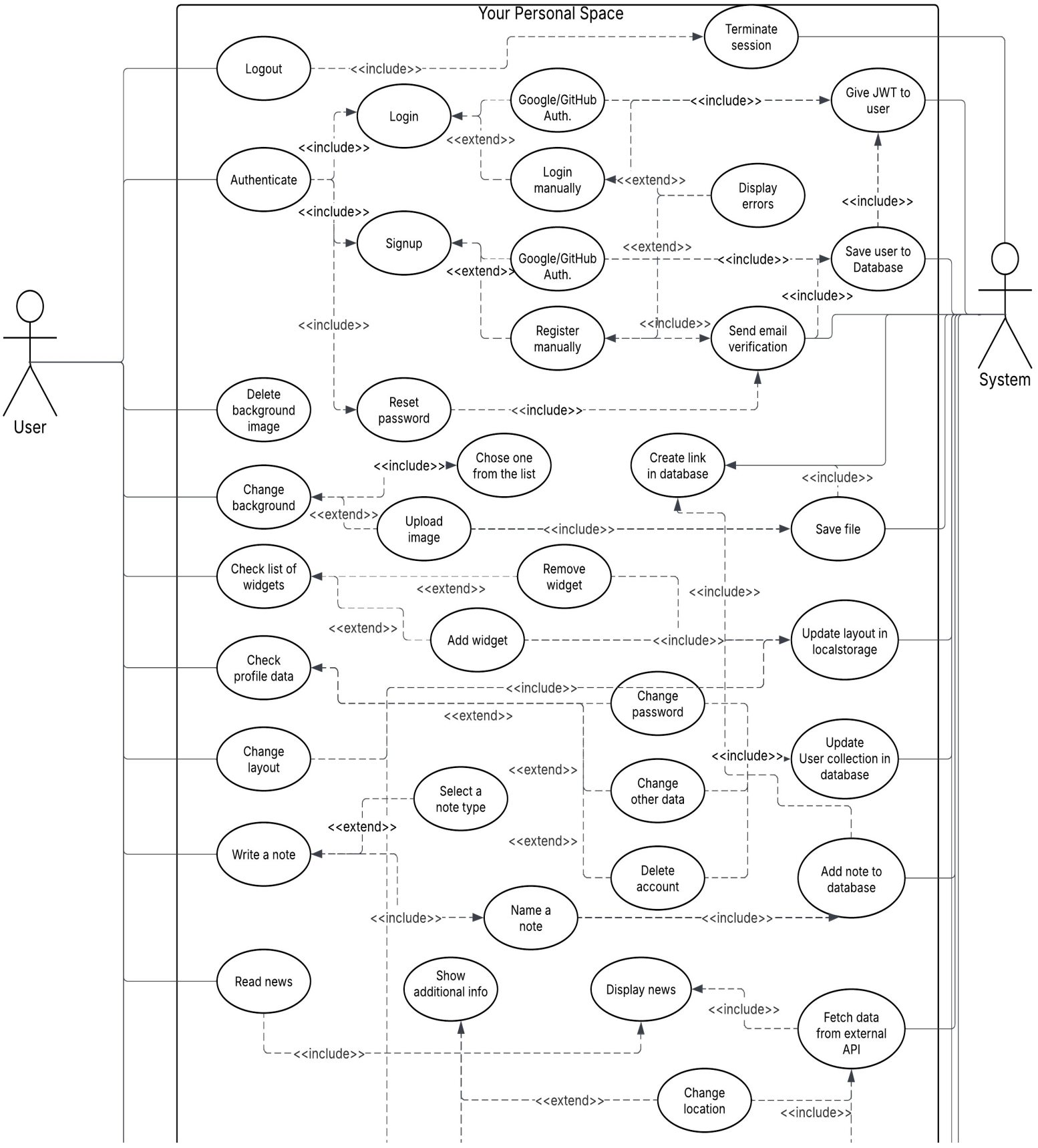


Figure 3.1: Use-case Diagram.1

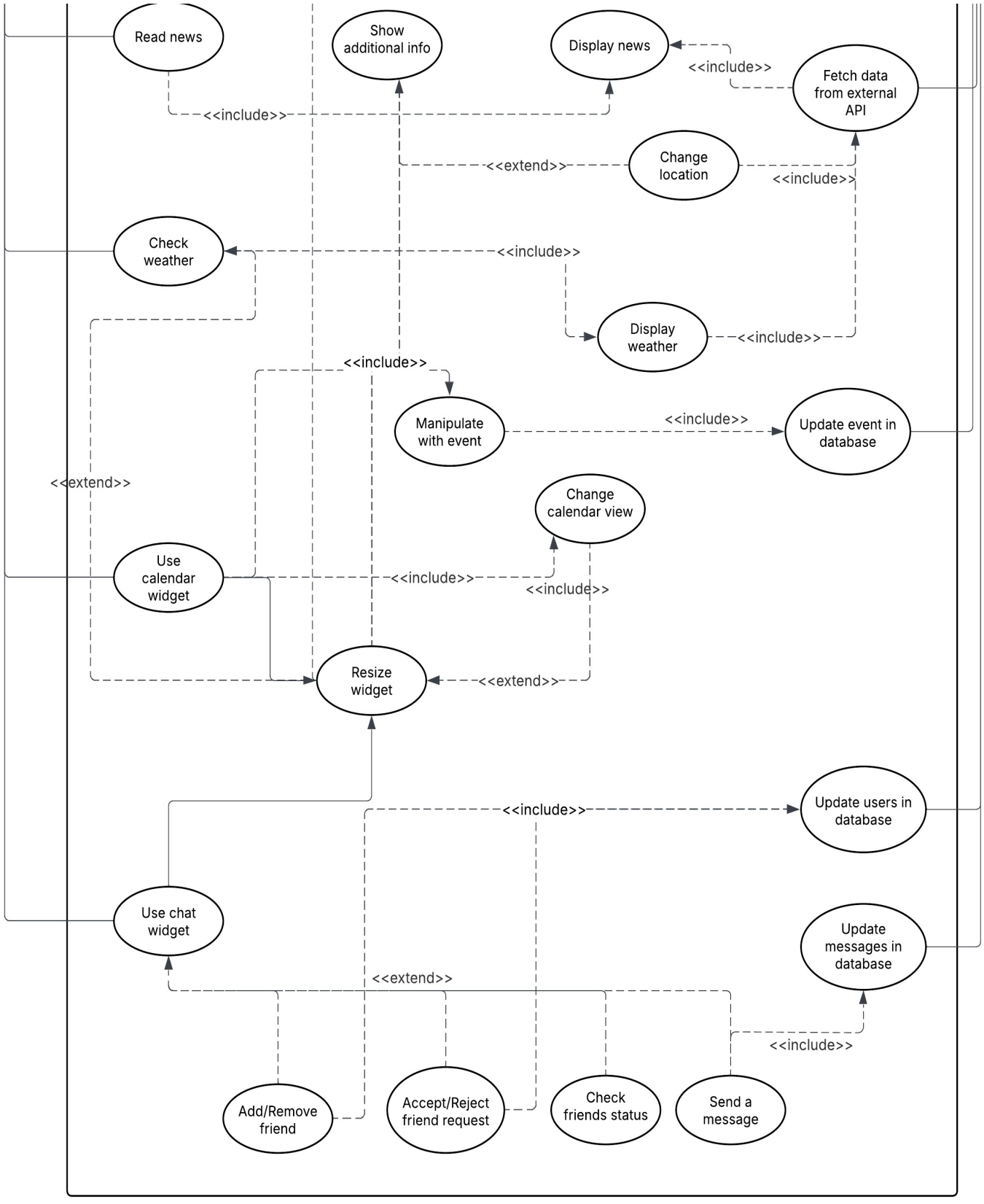


Figure 3.2: Use-case Diagram.2

3.2 Technologies

This project leverages a modern and comprehensive technology stack to deliver a robust, scalable, and user-friendly web application. The stack is divided into frontend, backend technologies and database, external API sections, each chosen to maximize developer efficiency, maintainability, and performance. From interactive UI design and responsive layouts to secure backend services and real-time communication, each tool and library plays a crucial role in building the overall system and user experience.

3.2.1 Frontend Technologies

Main Technologies

Table 3.1: Frontend Main Technologies

Technology	**Description**
React 19	JavaScript library for building user interfaces, emphasizing component-based architecture.
TypeScript	A superset of JavaScript that adds static typing to improve developer productivity and code quality.
Vite	A fast build tool and development server that optimizes frontend development with instant hot module replacement.
Tailwind CSS	A utility-first CSS framework for rapidly building custom, responsive UIs.

Frontend Libraries

Table 3.2: Frontend Libraries and Tools

Library	Description
Material-UI (MUI)	React components implementing Google's Material Design for faster UI development.
Styled Components	A CSS-in-JS library that allows you to write actual CSS to style your components.
Emotion	A performant and flexible CSS-in-JS library that supports both styled components and style objects.
Framer Motion	A production-ready animation library for React to create smooth animations and transitions.
React Grid Layout	A responsive grid layout system for React that enables draggable and resizable widgets.
React Hooks	Built-in functions that let use state and other React features without writing a class.
React Router DOM	Declarative routing for React web applications.
Socket.IO Client	A JavaScript library for real-time communication between the browser and server.
date-fns	A modern JavaScript date utility library offering simple, consistent, and powerful functions.
uuid	A library used to generate RFC-compliant unique identifiers.
React Toastify	Allows for beautiful, customizable toast notifications in React apps.
Google Fonts (Montserrat, Poppins)	Stylish and modern typefaces used for clean and aesthetic typography.
ESLint / TypeScript ESLint	Linting tools for maintaining code quality and consistent formatting.
PostCSS / Autoprefixer	Post-processing tools for transforming CSS and automatically adding vendor prefixes.

3.2.2 Backend Technologies

Main Technologies

Table 3.3: Backend Main Technologies

Technology	Description
Node.js	A JavaScript runtime built on Chrome's V8 engine. Used with TypeScript to prevent compilation-time errors.
TypeScript	Enables strong typing in Node.js, enhancing maintainability and developer experience.
Express.js	A minimal and flexible Node.js web application framework for building RESTful APIs.

Backend Libraries

Table 3.4: Backend Libraries and Middleware

Library	Description
Passport.js	Middleware for handling authentication in Node.js applications.
JWT (jsonwebtoken)	Securely transmits information between parties using JSON Web Tokens.
bcrypt	Library to hash passwords securely.
GitHub OAuth (passport-github2)	Authentication strategy using GitHub accounts.
Google OAuth (passport-google-oauth20)	OAuth 2.0 strategy using Google accounts.
Socket.IO	Enables real-time, bidirectional communication between clients and servers.
Multer	Middleware for handling multipart/form-data, mainly used for file uploads.
Express-session	Middleware to manage session data on the server side.
Cookie-parser	Middleware to parse cookies attached to client requests.
Nodemailer	Module for sending emails easily in Node.js.
ts-node / ts-node-dev / nodemon	Tools for executing TypeScript and reloading the server during development.
dotenv	Loads environment variables from a ‘.env‘ file into ‘process.env‘.

3.2.3 Database

Table 3.5: Database

Technology	Description
MongoDB	Non-SQL document-based database used for storing data.
Mongoose	ODM (Object Data Modeling) library used in Node.js to interact with MongoDB.

3.2.4 External APIs

Table 3.6: External APIs

Service	Description
Visual Crossing Weather API	Provides weather data for any location globally.
NewsData.io API	Supplies real-time and historical news data from worldwide news sources.
OpenCage Geocoding API	Transforms geographic coordinates into readable location data and vice versa.

3.3 Installation

This section provides detailed instructions for setting up the development environment required to run the application. The application consists of two main parts: a backend built with Node.js and Express, and a frontend developed using React and Vite. Ensure all prerequisites are installed before proceeding.

Note: On macOS or Linux systems, `sudo` may be required for some commands, especially when installing global dependencies.

If the project is public, it can be cloned using Git. Otherwise, a manual download of the repository is required.

```
git clone https://github.com/shepelenkomikhail/YPS.git
```

Code 3.1: Cloning the Repository

3.3.1 Backend installation

```
cd backend
npm install
```

Code 3.2: Navigate to Backend and Install Dependencies

Create a ‘.env‘ file in the backend root directory and populate it with necessary environment variables.

```
    PORT=8000
    SECRET_KEY=your secret

    MONGO_URI=your db
```

```
    GOOGLE_CLIENT_ID=your secret
    GOOGLE_CLIENT_SECRET=your secret
    GITHUB_CLIENT_ID=your secret
    GITHUB_CLIENT_SECRET=your secret
    NEWSDATA_API_KEY=your secret
    WEATHER_API_KEY=your secret
    OPENCAGE_API_KEY=your secret
    EMAIL_USER=your secret
    EMAIL_PASS=your secret
```

Code 3.3: Backend .env File

3.3.2 Frontend installation

```
cd frontend
npm install
```

Code 3.4: Navigate to Frontend and Install Dependencies

3.3.3 Run

When you are in the Backend folder, run following comand to run server:

```
npm start
```

Code 3.5: Run Backend Server

When you are in the Frontend folder, run following comand to run client:

```
npm run dev
```

Code 3.6: Run Frontend Dev Server

Access the app via:

- Frontend: `http://localhost:5173`
- Backend: `http://localhost:8000`

3.4 Architecture

This section describes the architecture of the project in details. The architecture is well structured to maintain comfortable navigation and efficient communication between components.

3.4.1 Folder Structure

The YPS project has a well-structured folder system for efficient navigation between files during development. The project is divided into two main parts: **Backend** and **Frontend**. Here's the reformatted backend structure following the same style as your frontend structure:

3.4.2 Backend Structure

The **Backend** folder is organized into distinct directories to maintain a clear separation of concerns.

`src` folder

- `controllers` - Business logic for handling HTTP requests and responses
- `middleware` - Custom middleware functions
- `models` - MongoDB schemas using Mongoose
- `routes` - API route definitions
- `services` - Core business logic and external API integrations
- `types` - TypeScript interfaces and type definitions
- `uploads` - User-uploaded files storage
- `server.ts` - Application entry point

`middleware`

- Authentication and authorization (JWT verification)
- Error handling and logging

- CORS configuration
- Request validation
- Session management

`models`

- User model for authentication and profile data
- Chat and message models
- Widget configuration models
- Calendar event models

`routes`

- `authRoutes.ts` - Authentication routes
- `calendarRoutes.ts` - Calendar event management
- `chatRoutes.ts` - Real-time chat functionality
- `coordinatesRoutes.ts` - Geolocation services
- `friendRoutes.ts` - Friend management
- `newsRoutes.ts` - News data handling
- `noteRoutes.ts` - Note management
- `profileRoutes.ts` - User profile management
- `uploadRoutes.ts` - File upload handling
- `userRoutes.ts` - User management
- `weatherRoutes.ts` - Weather data handling

Root folder

`.env` Environment variables configuration

`.gitignore`
: Version control ignore rules

`package.json`
: Project configuration and dependencies

`tsconfig.json`
: TypeScript compiler configuration

`node_modules/`
: Installed npm packages

3.4.3 Frontend Structure

The **Frontend** folder is organized into distinct directories to maintain a clear separation of concerns.

`src` folder

- `assets` folder – styles and media content
- `components` – reusable UI components
- `context` – context provider
- `types` – types definition, input patterns, and map to match weather conditions with corresponding image from `assets`
- `pages` – divided into main pages: `auth_page` and `home_page`

`home_page`

- `navbar` – Header component and its elements divided into separate folders
- `utils` – utility functions
- `widgets` – `Desk` component that handles widgets and all widget implementations divided into folders

- `Home Page` component – container component

`auth_page`

- `callback` – callback page
- `components` – reusable UI components
- `login` – login page and password reset page
- `signup` – signup page and email confirmation page

Root folder

`package-lock.json`

Lock file for npm dependencies, ensuring consistent package versions across installations.

`package.json`

Project configuration file containing:

- Project metadata
- Dependencies list
- Development scripts
- Project configuration

`node_modules/`

Directory containing all installed npm packages and their dependencies.

`src/` Source code directory containing:

- React components
- Application logic
- Styles
- Configuration files

`index.html`

Main HTML template file that:

- Serves as the entry point

- Contains root DOM element
- Includes necessary meta tags

`eslint.config.js`

ESLint configuration file defining:

- Code style rules
- Best practices
- Error prevention

`tailwind.config.js`

Tailwind CSS configuration file containing:

- Theme customization
- Color palette
- Breakpoints
- Plugin configuration

`public/`

Directory for static assets:

- Images
- Fonts
- Other static files

`postcss.config.mjs`

PostCSS configuration file for:

- CSS processing
- Tailwind integration
- Autoprefixer setup

`vite.config.ts`

Vite bundler configuration file defining:

- Development server settings
- Build options

- Plugin configuration

`.gitignore`

Git configuration file specifying:

- Files to ignore in version control
- Build artifacts
- Environment files
- System files

`tsconfig.node.json`

TypeScript configuration for Node.js environment.

`tsconfig.app.json`

TypeScript configuration for application code.

`tsconfig.json`

Base TypeScript configuration file containing:

- Compiler options
- Type checking rules
- Module resolution

3.4.4 Diagrams

There are general diagrams of main components used in the project that show how application works.

Frontend Authentication Page

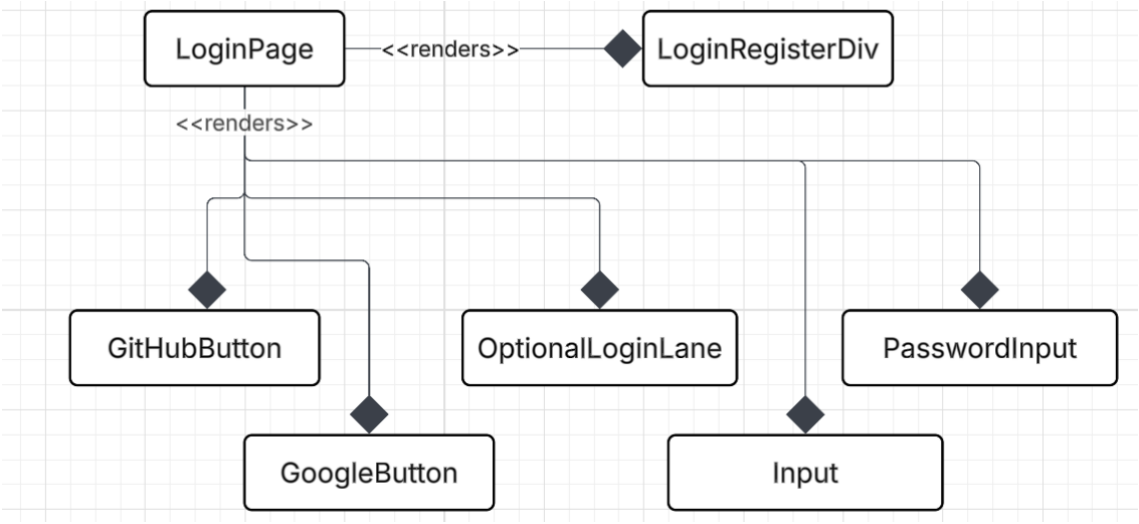


Figure 3.3: Login Page diagram

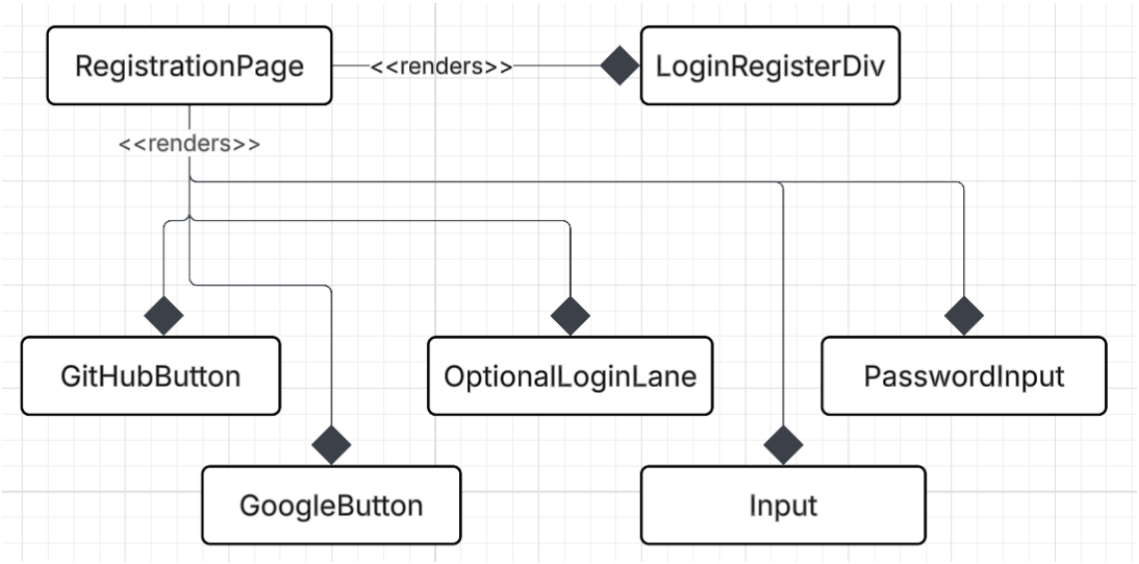


Figure 3.4: Signup Page diagram

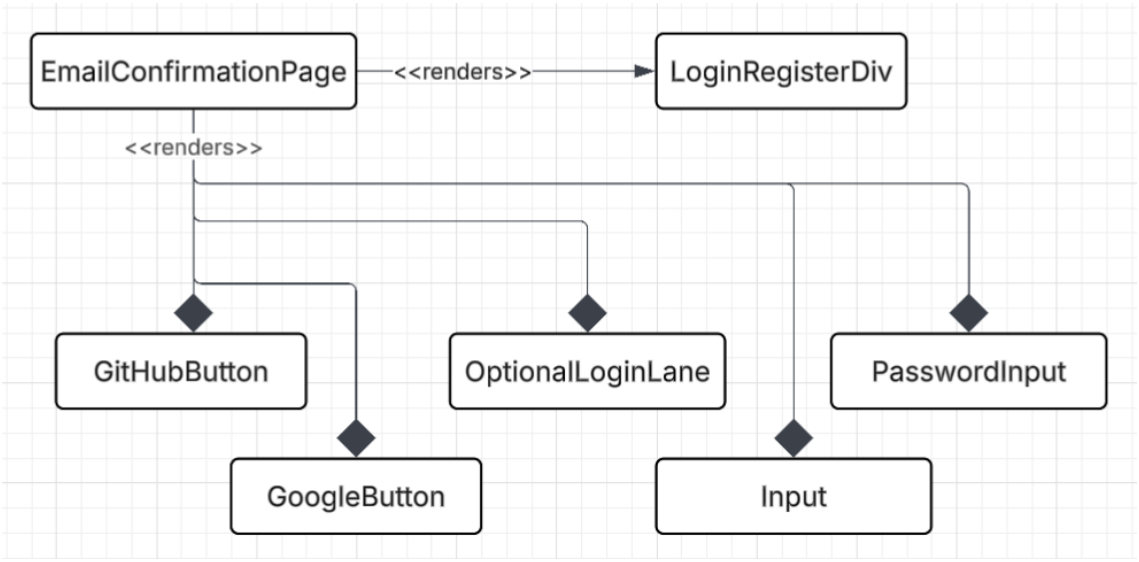


Figure 3.5: Email Confirmation Page diagram

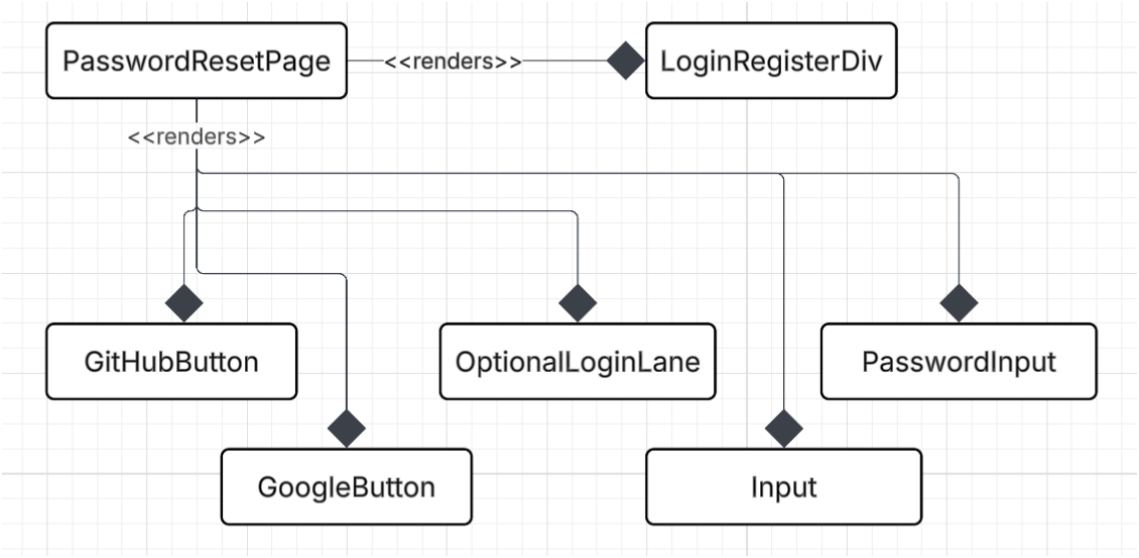


Figure 3.6: Password Reset Page diagram

Frontend Home Page

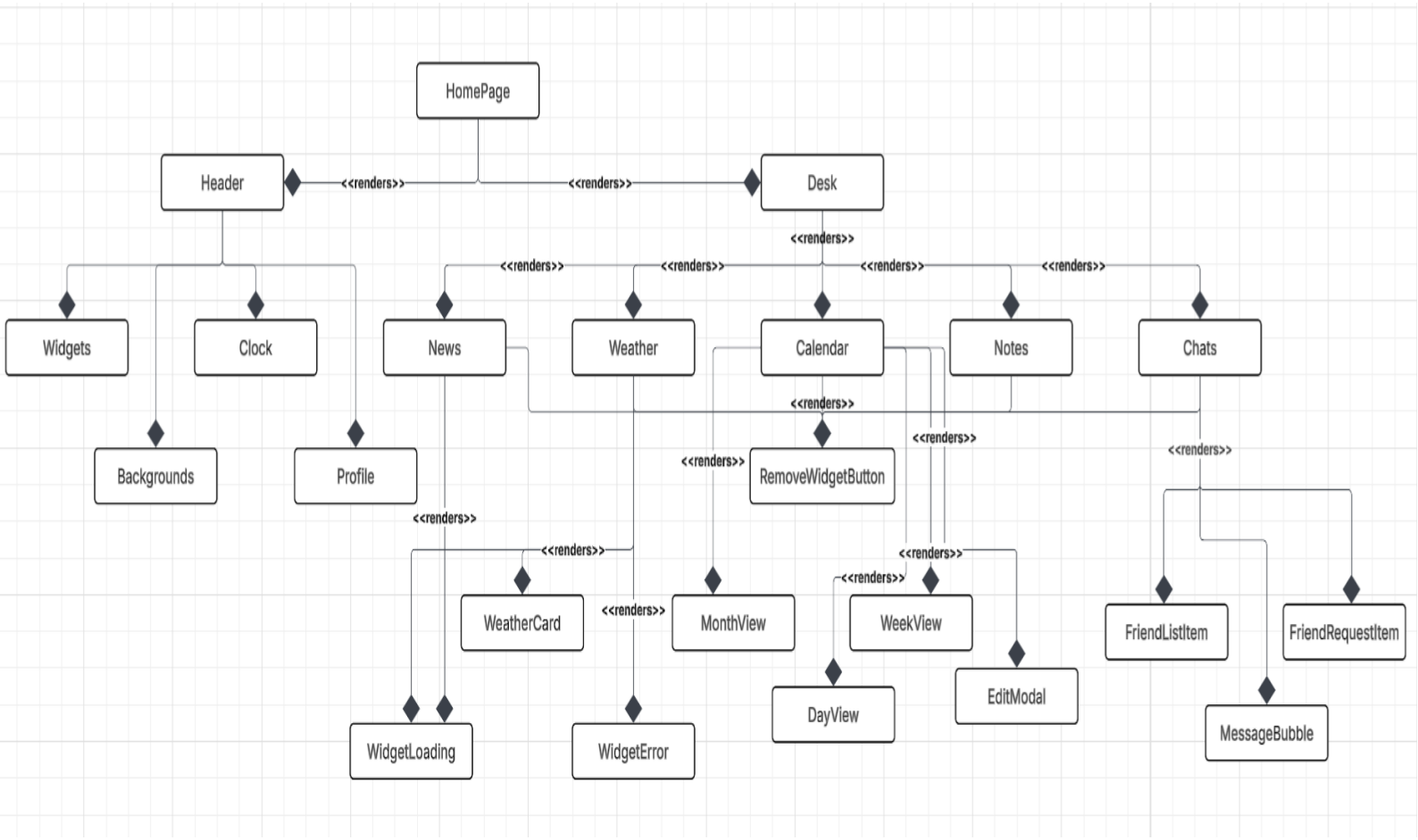


Figure 3.7: Home Page diagram

Backend Requests Flow

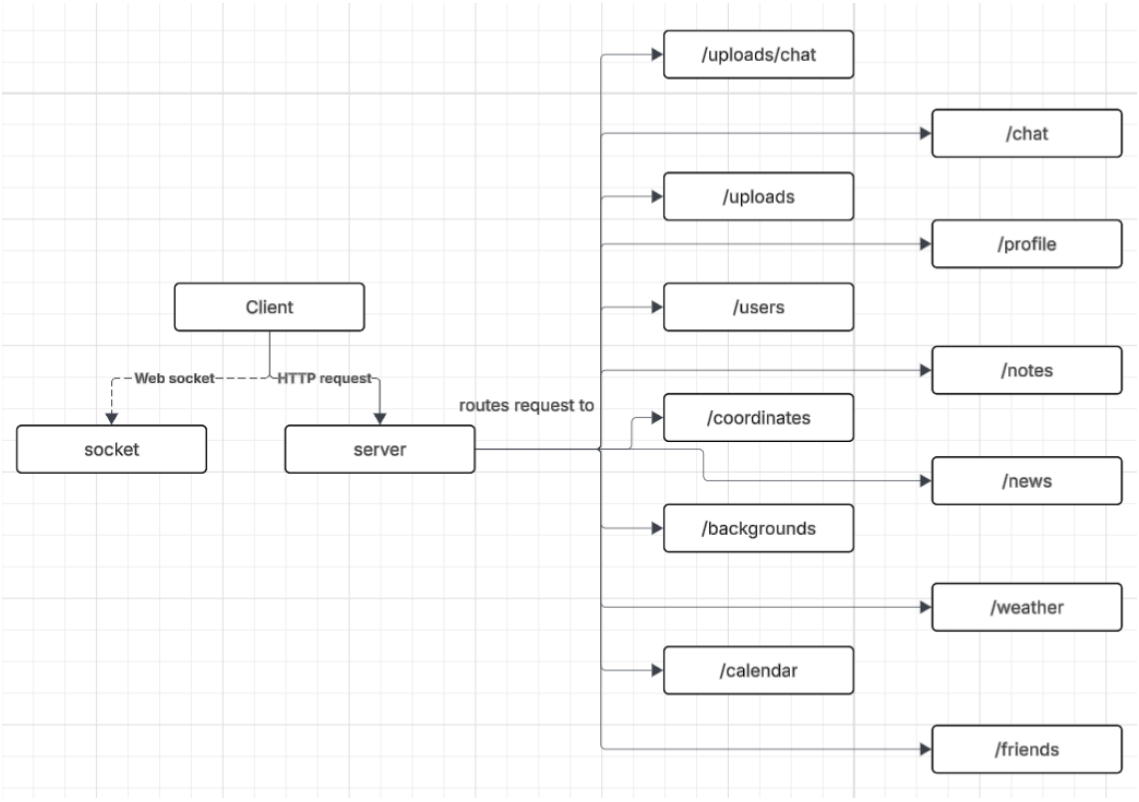


Figure 3.8: Requests flow diagram

3.4.5 Database Layer

Quality Description

The database layer implements user authentication through a sophisticated schema design in the UserModel, which includes unique indexes on username and email fields, password hashing with bcrypt, and OAuth integration fields (googleId, githubId). The login process performs a findOne query on the username field and uses bcrypt.compare for password verification.

Registration creates a new document with hashed password and verification fields, using MongoDB's unique constraint to prevent duplicate usernames or emails.

Password reset functionality utilizes temporary fields (tempPassword, resetCode, resetCodeExpires) with atomic updates through findOneAndUpdate to ensure data consistency.

File uploads are managed through two separate models: ImageSchema for profile images and ChatFileModel for chat attachments. The ImageSchema maintains a one-to-many relationship with users through the images array in UserModel, while

ChatFileModel stores file metadata with references to the uploading user. Both models implement timestamp tracking and use MongoDB's ObjectId references for efficient querying.

User profile updates are handled through atomic operations using findOneAndUpdate with validation checks for unique fields, and the images array is managed using MongoDB's array operators for efficient updates.

The calendar system is implemented through the EventModel, which maintains a one-to-many relationship with users through the user reference field. Events are stored with required fields (title, startDate, endDate) and optional fields (description, location), with timestamps for creation and updates. The model uses MongoDB's date type for efficient date-based queries and sorting.

Notes are managed through a separate schema that supports multiple content types (orderedList, checklist, text) with user references and timestamps for version tracking.

The friend system is implemented through a combination of the UserModel's friends array and the FriendRequestModel. Friend requests are stored as separate documents with sender and receiver references, status tracking, and creation timestamps. When a request is accepted, both users' friends arrays are updated atomically using MongoDB's $addToSet operator to prevent duplicates. The system uses compound indexes on sender and receiver fields for efficient querying of friend relationships.

User status management is implemented through the UserModel's status field with an enum constraint (online, away, offline) and is updated atomically using findOneAndUpdate. The system maintains an in-memory map of user statuses for real-time updates while persisting the data in MongoDB.

Message exchange is handled through the MessageModel, which stores messages with sender and receiver references, content, attachments array, and status tracking. The model uses MongoDB's default timestamp for message ordering and implements atomic updates for status changes.

File storage for chat implements security measures through the ChatFileModel, which stores file metadata including URL, original name, MIME type, and user reference. The model uses MongoDB's timestamp for tracking upload times and maintains references to users for efficient querying. Each operation in the system is implemented with proper indexing, atomic operations, and validation to ensure data integrity and performance.

The database layer uses MongoDB's aggregation framework for complex queries and implements proper error handling for database operations.

Data Operations

- **Create Operations:**
  - Uses Mongoose's `save()` method
  - Automatic timestamp generation
  - Reference population for related documents
  - Validation before save
- **Read Operations:**
  - `find()` for multiple documents
  - `findOne()` for single documents
  - Population of references (images, friends)
  - Query filtering and sorting
- **Update Operations:**
  - `findOneAndUpdate()` for atomic updates
  - Validation during update
  - Automatic timestamp updates
  - Reference integrity maintenance
- **Delete Operations:**
  - `findOneAndDelete()` for atomic deletion
  - Cascade handling for related documents
  - Reference cleanup
- **Data Validation:**
  - Schema-level validation
  - Required field checking
  - Unique constraints

  - Enum value validation
  - Reference existence verification

- **Indexing:**
  - Unique indexes on `username` and `email`
  - Timestamp indexes for sorting
  - Reference indexes on foreign keys

Quantity Description

Core Models

- `UserModel.ts` handles:
  - OAuth integration (Google and GitHub IDs)
  - Basic user information (`firstName`, `lastName`, `username`)
  - Email and password authentication
  - Image and friend references
  - User status tracking (`online/away/offline`)
  - Email verification system
  - Password reset functionality
- `MessageModel.ts` manages:
  - Basic message structure (`senderId`, `receiverId`)
  - Message content and attachments
  - Timestamp tracking
  - Message status (`sent/read`)
- `EventModel.ts` contains:
  - Basic event information (`title`, `description`)
  - Date ranges (`startDate`, `endDate`)
  - Optional location data
  - User reference

Supporting Models

- `FriendRequestModel.ts` tracks:
  - Sender and receiver references
  - Status (`pending/accepted/rejected`)
  - Creation timestamp
- `ChatFileModel.ts` stores:
  - File URL and original name
  - MIME type information
  - User reference
  - Upload timestamp
- `UserStatus.ts` maintains:
  - Basic status interface (`online/offline/away`)
  - User ID reference
  - Optional last seen timestamp
- `ImageSchema.ts` manages:
  - Image URL storage
  - User reference
  - Automatic timestamp tracking

Database Relationships

- **One-to-many:** `Users` → `Messages`, `Users` → `Events`
- **Many-to-many:** `Users` ↔ `Friends`
- **One-to-many:** `Messages` → `File attachments`
- **One-to-many:** `Users` → `Friend requests`

The database design balances normalization with practical performance considerations, ensuring efficient operations while maintaining data integrity. Schema validation occurs at both application and database levels, providing multiple safeguards against invalid data.

3.4.6 Client layer

Client layer is built using React components and main logical principles of React like props, hooks and reusable components. Virtual DOM that is offered by this framework ensures good user experience and smooth rerenders.

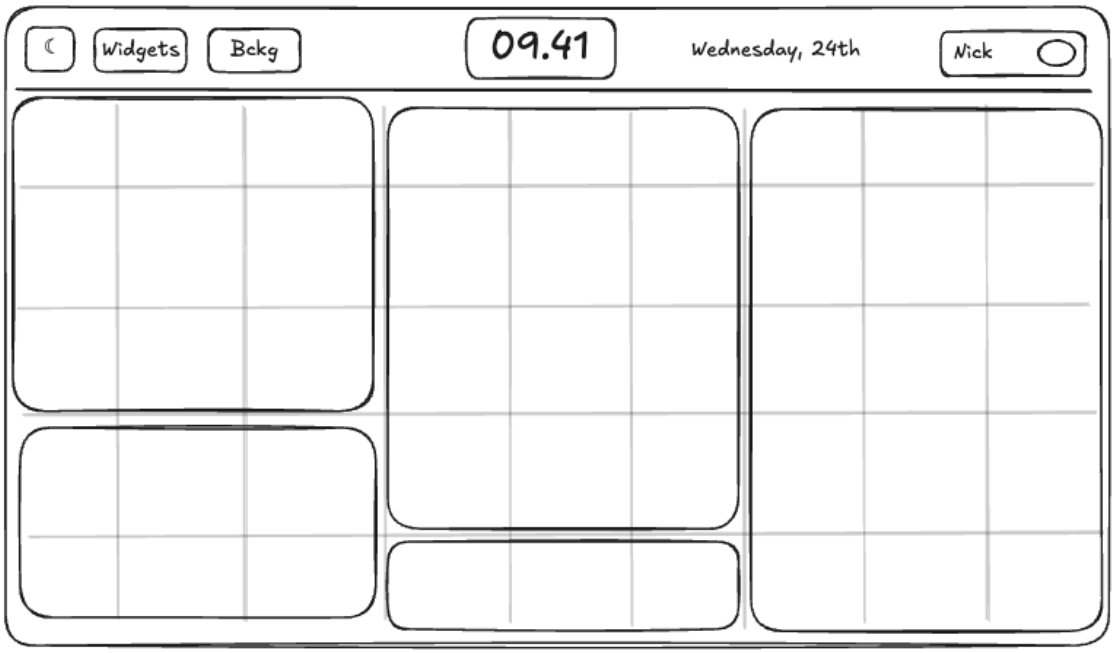


Figure 3.9: Design stage

At the design phase of the project client layer looked like that. The idea is kept, however some changes are made that improved UI and UX.

Quality description

Authentication page

Upon visiting the YPS application, users are greeted with the login page, which offers multiple authentication methods. The login form immediately checks the format of the entered data using pattern validation, such as `patternUsername` and `patternPassword`, and displays relevant error messages if the inputs do not meet the criteria. If the validation passes, a POST request is sent to the backend at `http://localhost:8000/users/login`. If the login is successful, the backend returns a JWT, stored securely in HTTP-only cookies with `SameSite` policy for CSRF protection and the `Secure` flag in production. The user is then redirected to

the home page. If the backend responds with an error, such as a 400 Bad Request or 500 Server Error, the appropriate error message is shown to the user, such as "Wrong username or password" or "Server Error," respectively.

The login page also integrates OAuth authentication through Google and GitHub. Clicking either button redirects the user to the respective service's login page. Upon successful authentication, the user is sent back to a callback page, which processes the response on the backend and redirects them to the home page. These OAuth flows are handled using reusable React components (`GoogleButton.tsx`, `GitHubButton.tsx`) and are visually organized with consistent styling using a shared container component (`LoginRegisterDiv.tsx`), along with support for dark mode and smooth UI transitions.

The same screen allows users to reset their password through a two-step process. Initially, the user is prompted to enter their email along with a new password and a confirmation field. The inputs are validated for correct formatting and password match. If the backend recognizes the email, it sends a code to the user's inbox. The form then transitions to a code confirmation view, handled within the same component. If the code is verified successfully through the backend, the user is redirected to the home page. Otherwise, an error message is shown. This entire process is managed through a credentials state in React, with loading states, error handling, and user feedback displayed using Material-UI's `Snackbar` and `Alert` components.

The signup page functions similarly to the login page but includes additional fields required for account creation. Like the login page, it includes robust form validation and makes a backend request upon submission. If all validations pass and the backend responds positively, an additional email confirmation component is displayed to finalize the signup process. This email confirmation ensures that only users with valid addresses can proceed.

Behind the scenes, all authentication-related components are built using React hooks for managing state, side effects, and navigation via React Router. Error and loading states are clearly communicated through the UI, which includes loading spinners and real-time feedback. Shared components like `PasswordInput.tsx` provide features such as password visibility toggles, while `OptionalLoginLane.tsx` offers visual separation between standard and OAuth login methods.

The entire authentication system emphasizes both user experience and security.

It ensures strong client-side validation for email formats, username patterns, and password rules, while server responses are handled gracefully with precise error messages. JWTs are stored using secure, modern cookie practices, and the overall interface provides a consistent, polished experience for both new and returning users.

Home page navbar

The header of the YPS application serves as the main navigation bar, designed with responsive behavior to accommodate both desktop and mobile screens. It includes elements such as the real-time clock, a theme toggle button, user profile access, widgets control, and background selection. On smaller screens, the header layout is simplified to display only a burger menu icon, the clock, and the theme toggle. The burger icon, when clicked, reveals a menu containing access to widgets, backgrounds, and user profile controls. This mobile behavior is enhanced with smooth animations for toggling the menu's visibility.

The clock component displays time in real-time using the browser's `Date` object, updating every second. It shows the full `HH:MM:SS` format on larger screens and a compact `HH:MM` format on smaller screens. For wider displays, it additionally shows the current weekday and date beneath the time. This component is optimized for performance, using interval updates and responsive design logic to adapt to different screen sizes.

The theme toggle button switches the visual theme of the page by dynamically modifying the class on the `body` element. It stores the user's preference in `localStorage` and communicates the selected mode through props to related components. The button features a custom animation defined in a `.css` file for smooth transitions between light and dark modes.

The widgets button provides access to a menu listing all available widgets with their respective icons. Widgets currently active on the dashboard display a remove icon next to their names. This menu supports interactions such as adding and removing widgets, with changes persisted to `localStorage` so the configuration is restored on subsequent page loads. The component handling this menu includes outside-click detection for automatic closing and implements smooth transition animations.

The backgrounds button allows users to customize the dashboard background. Clicking it opens a list of available background images. When a background is selected,

the application sets it and reloads the page to apply the change. Each background option includes a remove icon that sends a request to the backend to delete the image and uses Material-UI's `Snackbar` component to display the result of the request. The default background remains visible and selectable at all times. The interface also includes an image upload area built with MUI's file upload feature. Users can upload images, which are sent to the backend via `FormData` with file validation and feedback on success or failure.

The profile button opens a dropdown menu containing options for logout and settings. Logging out triggers an API request that clears the session on the backend and navigates the user back to the login page. Accessing the settings opens a modal that displays the user's profile data. Fields in this modal are initially read-only and can be made editable by clicking an edit icon. Clicking on the avatar icon allows the user to upload a new image. Below the form are buttons for deleting the account, changing the password, and saving changes. All actions are backed by API calls, with appropriate loading states, form validation, and error handling.

From a technical perspective, the header and its components are built using React and manage state with `useState` and related hooks. API operations use HTTP-only cookies for JWT-based authentication. Uploads are managed using authenticated `FormData` requests, with validation for image size and format. The application handles various error scenarios, including network failures, backend validation errors, and authorization issues, all surfaced to the user via Snackbars. Overall, the navbar provides an efficient, user-friendly, and secure interface for navigation and customization within the YPS application.

Home page dashboard

The YPS application's dashboard serves as the central hub for user interaction, providing a customizable workspace where various widgets can be arranged and managed. The dashboard is built using React Grid Layout, allowing for responsive and dynamic widget positioning. It maintains widget states and layouts in localStorage for persistence across sessions, while also adapting to different screen sizes through automatic layout adjustments. The dashboard supports both light and dark themes, with the selected theme being stored in localStorage and applied through dynamic class toggling on the body element.

The Notes widget provides a simple yet powerful interface for managing personal notes. It uses React's useState and useEffect hooks for state management and side effects. The widget makes authenticated API calls to the backend using fetch with proper headers and credentials. It implements optimistic UI updates for better user experience, showing changes immediately before confirming with the backend. Error handling includes retry mechanisms and user notifications through Material-UI Snackbar components. The widget uses controlled components for form inputs and implements debouncing for save operations to prevent excessive API calls.

The Weather widget displays current weather conditions and forecasts for the user's location. It uses the browser's Geolocation API to determine the user's position and makes authenticated requests to the weather API endpoint. The widget implements caching mechanisms to reduce API calls and uses React's useMemo for performance optimization of weather data processing. It includes error boundaries to handle API failures gracefully and provides fallback UI states. The widget updates at configurable intervals using setInterval with proper cleanup in useEffect.

The Calendar widget offers a comprehensive view of scheduled events and appointments. It uses a combination of React state and context for managing calendar data across components. The widget implements drag-and-drop functionality for event rescheduling and uses date-fns for date manipulation and formatting. It makes authenticated API calls to fetch and update calendar events, with proper error handling and loading states. The widget includes memoized components for performance optimization and implements virtual scrolling for handling large numbers of events.

The news widget implements a news delivery system that operates through multiple layers. When a user loads the widget, it initiates a request flow that starts with the frontend component (News.tsx). The widget first checks for cached data using a timestamp-based caching mechanism (300,000ms or 5 minutes). If the cache is invalid or empty, it sends a POST request to the backend endpoint http://localhost:8000/news/get-news with the user's country code in the request body. The request first passes through the backend's authentication middleware (verifyToken), which validates the user's JWT token from the HTTP-only cookie. If authentication succeeds, the request reaches the news route handler. The backend then validates the location parameter and checks for the required NewsData API key in the environment variables. It constructs a request to the external NewsData API (https://newsdata.io/api/1/news), including parameters for country-specific news,

image requirements, and English language filtering. The external API processes the request and returns a JSON response containing news articles. The backend then forwards this response to the frontend, maintaining the original data structure. The frontend widget processes this response through several steps: it validates the response for errors, checks for the presence of articles, and selects a random article using the getRandomArticle function. The selected article is stored in the component's state and displayed in a Material-UI Card component. The widget implements a display system that adapts to different screen sizes using the useResizeObserver hook. It renders the article's title, description, source name, and publication date in a responsive layout that switches between row and column orientations based on the container width. The widget also includes error handling that displays appropriate error messages and loading states using custom components (WidgetLoading and WidgetError). For continuous updates, the widget implements an automatic article rotation system that changes the displayed article every 5 minutes using a setInterval timer. This timer is properly cleaned up when the component unmounts to prevent memory leaks. The widget also includes a manual refresh capability through a refresh button that triggers a new API request.

The chat widget implements a real-time messaging system that operates through multiple interconnected layers. When a user initiates a chat interaction, the system follows a flow that begins with the frontend components and extends through various services to the backend. The communication flow starts with the useChatSocket hook, which establishes a WebSocket connection using Socket.IO. This hook manages the real-time communication layer, handling socket events for message sending and receiving. When a user sends a message, the chatService processes the request through its sendMessage function, which emits a 'send_message' event through the socket connection. The service includes error handling and promise-based response management, ensuring reliable message delivery. The backend receives these socket events through the Socket.IO server, which processes them in several steps. First, it validates the sender and receiver IDs, then processes the message content and any attachments. The message is saved to the database through the saveMessageToDatabase service, which creates a new message record with status 'sent'. The backend then emits the message to the appropriate chat room using the room ID generated from the sorted user IDs. For file attachments, the system implements a separate upload flow. When a user attaches a file, the chatService.uploadFile function

sends a POST request to the backend's upload endpoint. The backend processes this through multer middleware, which handles file storage and validation. The uploaded file is stored in the server's file system, and the file URL is returned to the frontend for inclusion in the message. Message display is handled by the MessageBubble component, which implements a sophisticated rendering system. It processes different types of content (text, images, videos, audio files) and displays them appropriately. The component includes animations using Framer Motion, status indicators for message delivery, and timestamp formatting. It also implements responsive design that adapts to different screen sizes and themes. The friends system implements a comprehensive social networking layer that manages friend relationships and friend requests through multiple interconnected components. The system begins with the friendService, which provides a complete API for friend management operations. When a user searches for potential friends, the service sends a GET request to /friends/search with the username query parameter. The backend processes this request by querying the UserModel for matching usernames, excluding the current user and existing friends, and returns the results with user profile information. Friend request management follows a sophisticated flow. When a user sends a friend request through friendService.sendFriendRequest, the system creates a new FriendRequest document in the database with 'pending' status. The backend then emits a real-time notification to the recipient through Socket.IO, triggering the new_friend_request event. The recipient's frontend receives this notification and updates the friend requests list through the FriendRequestItem component, which displays the request with the sender's profile information, including avatar and username. The friend request handling process involves several steps. When a user responds to a request through friendService.respondToRequest, the system updates the request status to either 'accepted' or 'rejected'. If accepted, the backend performs a two-way friend relationship update by adding each user to the other's friends list in the UserModel. The system then emits real-time updates to both users through Socket.IO, triggering the friend_request_updated event. The frontend updates both users' friend lists and request notifications accordingly. The FriendRequestItem component implements a UI for request management. It displays the sender's information with proper avatar handling (falling back to initials if no avatar is available) and includes accept/reject buttons with appropriate icons. The component implements proper event handling to prevent event propagation and includes loading states during request processing.

Friend status tracking is implemented through the friendService.fetchStatuses function, which retrieves the online/offline status of all friends. The backend maintains a Map of user statuses that is updated in real-time through Socket.IO connections. When a user's status changes, the system emits updates to all their friends through the friend_status event.

Each widget on the dashboard is implemented as a separate React component, with its own state management and API integration. The widgets communicate with the backend through authenticated API calls, using JWT tokens stored in HTTP-only cookies. The dashboard layout is responsive and adapts to different screen sizes, with widgets automatically resizing and repositioning to fit the available space. User preferences, including widget positions and active widgets, are persisted in localStorage to maintain the dashboard's state across sessions.

The dashboard's background can be customized through the backgrounds menu, with options to upload new images or select from existing ones. The selected background is stored in localStorage and applied to the dashboard, creating a personalized workspace for each user. The entire dashboard is designed with a focus on user experience, providing smooth animations, intuitive interactions, and consistent styling across all components.

Quantity description

Authentication Page – Quantity Description

The authentication system in the YPS application is implemented as a collection of React components and supporting utilities, each managing a specific piece of the login, registration, password reset, and OAuth functionality. The primary components include `LoginPage.tsx`, `PasswordResetPage.tsx`, `RegistrationPage.tsx`, `EmailConfirmationPage.tsx`, and `CallbackPage.tsx`, with each relying heavily on React's state and effect hooks for managing application logic, user interaction, and side effects.

Within `LoginPage.tsx`, multiple `useState` hooks are used to handle form fields like username and password, as well as internal states for tracking loading indicators, error messages, and dark mode preferences. A `useEffect` hook captures and processes error messages from URL parameters to provide immediate feedback if an OAuth login fails. The component defines key functions such as `handleSubmit`, which handles the

login form submission logic, `signupNavigation` for routing users to the registration page, and `resetPassword` for navigating to the password recovery page. It interacts with the backend by sending a POST request to `/users/login` to authenticate users.

`PasswordResetPage.tsx` manages a two-step password reset flow using React state to capture user inputs such as email, new password, confirmation password, and the verification code sent via email. The component defines internal credential state and tracks loading and error feedback using state variables. It provides two core functions: `handleSubmit`, which initiates the password reset by validating input and sending a request to `/users/verifypassword`, and `handleCodeCheck`, which verifies the code against the backend at `/users/verifypasswordreset`. Navigation back to the login screen is handled by `loginNavigation`. All feedback, including success and failure notifications, is communicated through Material-UI's `Snackbar` and `Alert` components.

In `RegistrationPage.tsx`, form fields such as username, email, and password are managed with local state, while separate validation states ensure input correctness before sending data to the backend. This component includes a `handleSubmit` function for form processing and a navigation handler to direct users back to the login screen. The registration API call is performed via a POST request to `/users/register`. Upon successful registration and validation, users are directed to the email confirmation process.

`EmailConfirmationPage.tsx` handles the verification of email codes through a controlled form interface. It stores the input code in local state and uses a function called `handleSubmit` to send the code to the backend via a POST request to `/users/verifyemail`. Successful verification grants users access to the application.

A number of shared components enhance consistency and reusability across the authentication system. `PasswordInput.tsx` offers a password field with a visibility toggle, real-time validation, and error feedback. The `LoginRegisterDiv.tsx` component wraps auth forms in a styled container, supporting both dark mode and responsive design. OAuth login is supported through `GoogleButton.tsx` and `GitHubButton.tsx`, which accept a dark mode prop and provide clickable buttons with customized styling and redirect logic. `OptionalLoginLane.tsx` adds visual separation between standard login and OAuth options.

`CallbackPage.tsx` processes the OAuth login flow upon redirect from the authentication provider. It uses a loading state to track progress and a `useEffect`

hook to handle the OAuth response automatically when the component loads. The `handleCallback` function extracts and verifies information from the URL and finalizes the login process. This component supports both Google and GitHub callbacks, interacting with `/auth/google/callback` and `/auth/github/callback` respectively.

Route protection is managed through the `ProtectedRoute.tsx` component, which wraps protected areas of the application. It checks the user's authentication state and, if necessary, redirects them to the login page. JWTs are stored as HTTP-only cookies, with CSRF protection enforced using the `SameSite` cookie policy and the `Secure` flag in production environments.

Validation throughout the authentication flow is enforced through custom regex patterns such as `patternUsername` and `patternEmail`, as well as password rules that require a minimum length, special characters, numeric digits, and uppercase letters. These patterns prevent invalid data from being submitted and provide immediate feedback within the UI.

Across all authentication components, state is managed through React's `useState` for local data and status tracking, and `useEffect` for reacting to external changes such as API responses or URL parameters. Events such as form submissions, input changes, button clicks, and OAuth redirects are tied to handler functions.

The UI leverages a combination of Material-UI components—such as `Button`, `TextField`, `Snackbar`, `Alert`, and `CircularProgress`—alongside Tailwind CSS classes for layout and styling. Responsive design ensures compatibility with all screen sizes, while animation classes provide smooth transitions between states.

Home Page Dashboard – Quantity Description

The dashboard view of the YPS application is orchestrated through `HomePage.tsx`, which acts as the central controller for layout configuration, widget management, background settings, and theming. It utilizes multiple state hooks to manage the current theme (`darkMode`), the list of active widgets (`activeWidgets`), the layout for the widget grid (`layout`), the current background image (`bgImage`), and the screen width for responsive design (`screenWidth`). A series of `useEffect` hooks ensure persistence of widgets and layout to `localStorage`, update the background image dynamically, and listen for window resize events. Key functions include `handleAddWidget` for injecting widgets into the dashboard, `removeWidget` for widget removal, and `handleTheme` for toggling between light and dark modes. The component receives props such as

`widgetConfig` to initialize widget layouts and `headerHeight` to calculate available vertical space beneath the navigation bar.

The `Desk.tsx` component serves as the interactive canvas for displaying widgets in a drag-and-drop grid layout. It manages a `mounted` state to control rendering behavior. Props passed to it include `headerHeight`, `activeWidgets`, `layout`, `setLayout`, and `removeWidget`, ensuring proper coordination with the parent component. Layout behavior is configured via parameters like `cols, rows` (to define the number of columns, rows per screen size), `rowHeight`, `margin`, and `containerPadding` to control the grid's structure.

Each widget is encapsulated in its own component. The `Notes.tsx` widget manages user-created notes and features a full CRUD interface. It maintains local state for the notes array, a loading flag, and a controlled input for new note entries. Notes are fetched and auto-saved using `useEffect` hooks, with debouncing applied for efficiency. Functions like `handleAddNote`, `handleUpdateNote`, and `handleDeleteNote` interact with API endpoints such as `GET /notes`, `POST /notes`, `PUT /notes/:id`, and `DELETE /notes/:id`.

The `Weather.tsx` widget handles real-time weather updates based on user location. It tracks weather data, loading status, and the current location using state hooks. Multiple `useEffect` hooks orchestrate the retrieval of geolocation data, fetch weather information, and periodically refresh data. Core logic is implemented in `fetchWeather`, which retrieves forecast data from `GET /weather` and `GET /weather/forecast`, while `handleLocationChange` enables users to search for different locations.

The `Calendar.tsx` widget provides a lightweight event management interface. State includes an array of scheduled events, the selected date, and a loading flag. Event fetching and date synchronization are handled via `useEffect`. Functions such as `handleAddEvent`, `handleUpdateEvent`, and `handleDeleteEvent` integrate with backend services through endpoints like `GET /events`, `POST /events`, `PUT /events/:id`, and `DELETE /events/:id`.

The `News.tsx` widget offers an infinite-scroll news feed. It tracks news articles, loading state, and the current page of results. The `useEffect` hooks manage fetching articles on component mount and handle infinite scrolling behavior. Core functions include `fetchArticles` for retrieving articles from `GET /news` and `handleLoadMore` to trigger pagination. Additional categories may be retrieved from

`GET /news/categories`.

`Chats.tsx` - the main chat component implements a real-time messaging system with friend management capabilities. The component maintains multiple state variables including friends list, selected friend, messages, and UI states. It implements WebSocket communication through Socket.IO for real-time messaging, typing indicators, and message read status updates. The component handles file attachments with size limits (15MB per file, 60MB total) and implements proper error handling and user feedback through toast notifications. The chat interface includes friend search functionality, friend request management, and message history loading with proper scroll behavior. The component uses React's useEffect hooks for managing socket connections, message synchronization, and UI updates. It implements proper cleanup of socket connections and event listeners to prevent memory leaks. `MessageBubble.tsx` The message bubble component renders individual chat messages with support for text content and file attachments. It implements a responsive design that adapts to both light and dark themes. The component handles different file types (images, videos, audio, documents) with appropriate rendering and download options. Each message displays a timestamp and read status indicators for sent messages. The component uses Framer Motion for smooth animations and implements proper event propagation handling. It includes avatar display for received messages and maintains consistent styling across different message types. `chatService.ts` The chat service implements core messaging functionality through API calls. It provides methods for sending messages, uploading files, and managing message status. The service handles proper error handling and response validation. It implements file upload functionality with proper MIME type handling and size validation. The service maintains consistent error handling patterns and provides clear feedback for failed operations. `friendService.ts` The friend service manages friend-related operations through API calls. It implements friend search functionality, friend request management, and friend status updates. The service handles proper error handling and response validation. It provides methods for sending friend requests, responding to requests, and managing friend relationships. The service maintains consistent error handling patterns and provides clear feedback for failed operations. `useChatSocket.ts` The chat socket hook manages WebSocket connections and event handling. It implements proper connection management and event listener setup. The hook handles message reception, typing indicators, and connection status updates. It

provides proper cleanup of socket connections and event listeners. The hook maintains consistent error handling patterns and provides clear feedback for connection issues. `validateRequest.ts` The validation service implements request validation for friend operations. It provides type checking and validation for friend requests and updates. The service maintains consistent validation patterns and provides clear feedback for invalid requests. It handles proper error handling and response validation. `getFileType.ts` The file type utility determines the type of file based on its extension. It implements proper MIME type detection and categorization. The utility handles common file types and provides appropriate categorization for display and handling. It maintains consistent file type detection patterns and provides clear feedback for unknown file types. `SearchResultItem.tsx` The search result component renders individual search results for friend search. It implements a responsive design that adapts to both light and dark themes. The component displays user information and provides action buttons for friend requests. It maintains consistent styling and behavior across different search results. `FriendRequestItem.tsx` The friend request component renders individual friend requests. It implements a responsive design that adapts to both light and dark themes. The component displays request information and provides action buttons for accepting or rejecting requests. It maintains consistent styling and behavior across different friend requests.

Loading and error states are tracked to improve user experience, with fallback mechanisms in place where appropriate. API integrations utilize `fetch` with credential inclusion, wrapped in try-catch blocks for error handling and recovery.

Performance is optimized using debouncing (e.g., for input fields), memoization, lazy loading of components, and virtual scrolling for content-heavy views like news and chat. Real-time updates are facilitated via WebSocket connections, with polling fallbacks and synchronized state for consistent UI behavior. UI elements are built using Material-UI components and styled with Tailwind CSS, incorporating responsive layouts and dark mode support.

Cache management and state restoration further enhance the user experience in disrupted or low-connectivity scenarios. Security considerations include input validation, XSS prevention, CSRF protection, and the use of secure API call patterns.

Event handling across the dashboard is supporting click interactions, form submissions, keyboard and touch events to have accessibility and usability across devices. Error boundaries and user feedback mechanisms ensure graceful degradation and

clear communication during unexpected issues.

3.4.7 Server layer

Quality description

Initial Request Handling

When a request arrives at the server, it first undergoes parsing and middleware processing. The request is initially parsed through a cookie parser, followed by JSON body parsing for non-Socket.IO requests. CORS headers are then configured using the setCorsHeaders middleware, and the request is logged using the logger middleware.

Next, the server performs authentication checks. Passport.js is initialized to manage authentication mechanisms, while session management is handled via the express-session module. For protected routes, a JWT token is verified, and if the request is successfully authenticated, user information is attached to the request object for downstream use.

Route Processing

After authentication, the request is routed to the appropriate handler. Routes are organized by feature modules such as authentication, users, and calendar. Each module defines specific API endpoints, with protected routes enforced by dedicated authentication middleware.

Within the controller layer, handlers receive the request and extract relevant data. Input validation is performed to ensure integrity, and the necessary business logic is invoked through service layer calls. Database operations are then executed using Mongoose models, completing the processing for that route.

Data Operations

The backend's data operations are managed across several layers. The service layer encapsulates business logic, performing complex operations, data transformations, and coordination of database interactions. It also handles validation and error management.

For actual database interaction, Mongoose is used as the ODM for MongoDB. Data schemas and validation rules are defined in models, and all CRUD operations

are carried out using model methods. The system also manages database transactions and includes comprehensive error handling during these operations.

Response Generation

Once processing is complete, the server prepares and sends a response. The response data is formatted according to API specifications, and appropriate status codes are applied based on the outcome of the operation. In the event of an error, descriptive and sanitized error messages are generated. Headers are also set to enhance security and manage caching.

Responses are primarily sent in JSON format for API endpoints. For file-related requests, file streams are provided, and real-time events are emitted using WebSocket protocols where applicable. Error responses are structured with appropriate HTTP status codes and secure messaging.

Real-time Communication

The backend supports real-time functionality via Socket.IO, which is initialized alongside the HTTP server. It handles client connection events and facilitates the emission of real-time events to connected clients, including support for room-based communication.

Socket events are managed through designated event handlers. Each event includes an authentication check to ensure secure interactions. Real-time updates are then broadcast to relevant clients, and error handling is implemented to manage any socket-related failures.

Error Handling

The backend includes error handling system that captures issues at multiple layers. Errors are appropriately formatted, assigned status codes, and logged for diagnosis. In production environments, stack traces are hidden to prevent sensitive data exposure.

Client-side errors result in 4xx status codes, while server-side errors yield 5xx codes. Error messages returned to the client are sanitized for security while maintaining informative descriptions to aid debugging.

Security Measures

Security is deeply integrated into the backend. Authentication mechanisms include JWT-based strategies, session handling, password hashing, and token expiration management.

Authorization is enforced through route protection, role-based access control, resource ownership checks, and API key validation.

To protect user data, the system performs strict input validation and implements security techniques to guard against SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF) attacks.

Quantity description

Middleware

`authMiddleware.ts` Implements two core functions for JWT token management. The `generateToken` function creates JSON Web Tokens using `userId` and `username`, signing them with a secret key and setting a 12-hour expiration. The `verifyToken` middleware extracts the token from cookies or headers, verifies it, and attaches decoded user information to the request. It returns a 403 status if no token is found and 401 if invalid.

`commonMiddleware.ts` Contains two request processing functions. The `setCorsHeaders` middleware sets CORS headers: `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods`, and `Access-Control-Allow-Headers`. The `logger` middleware logs the HTTP method and URL of incoming requests to the console.

`passportConfig.ts` Configures two OAuth strategies (Google and GitHub) and session serialization. The Google strategy uses client credentials from environment variables to authenticate users, verify email, and handle user creation or update. The GitHub strategy performs similar steps with additional email and profile parsing logic. Includes `serializeUser` and `deserializeUser` for session management.

`multerConfig.ts` Configures file upload handling with two storage options. The main configuration allows only image files (JPEG, PNG, GIF, WEBP), uses disk storage with unique filenames, and limits size to 15MB. A separate configuration

is used for chat uploads. Both setups support single-file uploads and export respective middleware.

`fileTypeFilter.ts` Implements file upload security. Defines a list of dangerous MIME types (scripts, executables, etc.). The `checkFileType` middleware deletes any matching uploaded files and returns a 403 status. Continues only if the file type is safe or no file is uploaded.

Controllers

`uploadControllers.ts` Handles file upload logic through four main functions:

- `uploadImage`: Handles profile/background image uploads. Verifies user authentication, processes the file, saves metadata to the database, and returns the file URL. Includes error handling for missing files or unauthorized access.
- `getImages`: Retrieves all images uploaded by the authenticated user. Queries the database and returns image URLs. Handles unauthorized access and server errors.
- `deleteImage`: Deletes images from both filesystem and database. Validates the request, removes the physical file and the corresponding database record. Handles edge cases like file not found, unauthorized access, or deletion errors.
- `uploadChatFile`: Processes chat-related file uploads. Authenticates the user, stores metadata (original filename, MIME type, timestamp), and returns detailed file info. Includes error handling.

Services

socket.ts Serves as the primary real-time communication service, implementing WebSocket functionality through Socket.IO. It manages user connections, status updates, and chat interactions. The service maintains two `Map` objects: `userSockets` for tracking active socket connections and `userStatuses` for monitoring user presence states. The `initializeSocket` function sets up the Socket.IO server with CORS configuration and handles various socket events including connection, disconnection, status updates, chat room management,

typing indicators, message sending, and read receipts. It implements a sophisticated room-based system for private messaging and status notifications, with automatic status updates and friend notifications.

saveMessage.ts Provides a focused service for persisting chat messages to the database. It accepts a message object containing `senderId`, `receiverId`, `content`, and optional `attachments`, creates a new `MessageModel` instance, and saves it to the database with a default `'sent'` status. The service includes logging for message content and returns the saved message object.

setRead.ts Handles message read status updates. It accepts an array of message IDs, retrieves the corresponding messages from the database, updates their status to `'read'` if not already marked as such, and saves the changes. The service uses `Promise.all` for efficient batch processing of message updates.

Types

express.d.ts This file extends Express.js's `Request` interface by adding an optional `user` property, which is imported from the `UserType` definition. This extension allows for type-safe access to user information throughout the request handling pipeline.

UserType.ts This file defines the core user data structure with properties including:

- `id`: Unique identifier
- `firstName`: User's first name
- `lastName`: User's last name
- `username`: User's username
- `email`: User's email address
- `password`: Authentication data
- `googleId`: Optional OAuth identifier

This type definition ensures a consistent user data structure across the application.

Note.ts This is the most comprehensive file, defining both TypeScript types and a Mongoose schema for note functionality. It includes:

- `NoteType` union type: `'orderedList' | 'checklist' | 'text'`
- `INote` interface extending Mongoose's `Document` type
- Properties for content, type, user reference, and timestamps
- Mongoose schema (`NoteSchema`) with required fields
- Exported `NoteModel` for database operations

Models

The `models` folder in the backend contains Mongoose schemas that define the data structure and relationships for the application.

UserModel.ts Serves as the core model, defining user properties including authentication fields (`googleId`, `githubId`, `password`), personal information (`firstName`, `lastName`, `username`, `email`), social features (`friends` array), and account management fields (`verificationCodes`, `resetCodes`). It includes a `status` field that tracks user presence (`'online'`, `'away'`, `'offline'`) and maintains relationships with images through `ObjectId` references.

UserStatus.ts Provides a lightweight interface for tracking user presence states, containing `userId`, `status`, and `lastSeen` timestamp. This model works in conjunction with the `UserModel` to provide real-time presence information across the application.

MessageModel.ts Handles chat communication, defining message structure with `senderId`, `receiverId`, `content`, `attachments` array, `timestamp`, and message status (`'sent'` or `'read'`). This model supports both text-based communication and file sharing through the `attachments` field.

ChatFileModel.ts Manages file uploads within chat conversations, storing file metadata including `URL`, original filename, MIME type, and upload timestamp. It maintains a reference to the uploading user through `userId`.

FriendRequestModel.ts Implements the social networking aspect, tracking friend requests between users. It includes `sender` and `receiver` references, request status (`'pending'`, `'accepted'`, `'rejected'`), and a creation timestamp.

EventModel.ts Handles calendar events, storing event details such as `title`, `description`, `start` and `end` dates, `location`, and `user` reference. It includes timestamps for creation and updates.

ImageSchema.ts Manages user-uploaded images, storing image URLs and maintaining user ownership through `userId` references.

Routes

userRoutes.ts Implements user management functionality through several key endpoints. The registration endpoint (`POST /`) handles user creation with email verification, implementing password hashing, verification code generation, and email sending. The verification endpoint (`POST /verification`) validates email verification codes and manages account activation. Password management is handled through two endpoints: `POST /verifypassword` generates and sends reset codes, while `POST /verifypasswordreset` validates and processes password resets. The user update endpoint (`PUT /:id`) manages profile updates including image uploads, username/email uniqueness validation, and password changes. Each endpoint includes proper error handling, input validation, and security measures.

friendRoutes.ts Manages social networking features through multiple endpoints. The friend list endpoint (`GET /`) retrieves a user's friends with their status and profile images. User search functionality (`GET /search`) implements case-insensitive username search with proper filtering. Friend request management includes sending requests (`POST /requests`), retrieving pending requests (`GET /requests`), and processing requests (`PUT /requests/:requestId`). The status endpoint (`GET /statuses`) provides real-time friend status updates, while the friend removal endpoint (`DELETE /:friendId`) handles friend list management. All endpoints implement authorization checks and real-time notifications through Socket.IO.

authRoutes.ts Implements authentication through multiple providers. The Google authentication flow includes two endpoints: `GET /auth/google` initiates the OAuth flow, while `GET /auth/google/callback` handles the authentication response. Similarly, GitHub authentication is implemented through `GET`

`/auth/github` and `GET /auth/github/callback`. The user verification endpoint (`GET /auth/me`) provides authenticated user information. Each endpoint includes token generation, cookie management, and error handling.

chatRoutes.ts Provides chat functionality through two main endpoints. The message history endpoint (`GET /`) retrieves chat messages with pagination and filtering. The message sending endpoint (`POST /`) handles message creation and real-time delivery through Socket.IO.

noteRoutes.ts Implements note management through CRUD operations. The note creation endpoint (`POST /`) handles new note creation with validation. The retrieval endpoint (`GET /`) provides note listing with optional filtering. The update endpoint (`PUT /:id`) manages note modifications, while the deletion endpoint (`DELETE /:id`) handles note removal. Each endpoint includes authorization checks and error handling.

calendarRoutes.ts Manages calendar events through several endpoints. The event creation endpoint (`POST /`) handles new event creation with date validation. The retrieval endpoint (`GET /`) provides event listing with date range filtering. The update endpoint (`PUT /:id`) manages event modifications, while the deletion endpoint (`DELETE /:id`) handles event removal. All endpoints include date validation and authorization.

uploadRoutes.ts Provides file upload functionality through a single endpoint (`POST /`). It handles file uploads with validation, storage management, and error handling, integrating with `multer` middleware for file processing.

profileRoutes.ts Manages profile operations through two endpoints. The profile retrieval endpoint (`GET /`) provides user profile information with proper data population. The update endpoint (`PUT /`) handles profile modifications with validation and authorization.

weatherRoutes.ts Provides weather data through two endpoints. The current weather endpoint (`GET /current`) retrieves current conditions, while the forecast endpoint (`GET /forecast`) provides predictions. Both include error handling and data formatting.

newsRoutes.ts Implements news retrieval through a single endpoint (`GET /`). It provides news articles with filtering, pagination, and error handling.

coordinatesRoutes.ts Manages location-based services through two endpoints. The coordinate conversion endpoint (`POST /convert`) handles coordinate transformations, while the location search endpoint (`GET /search`) provides location-based searches. Both endpoints include validation and error handling.

server.ts

The `server.ts` file serves as the main entry point for the backend application, implementing server initialization, middleware configuration, and route setup. It defines the overall structure and integration of the backend system.

Server Configuration Sets up environment variables and initializes the Express application. It defines critical constants such as the server port (`PORT`), MongoDB URI (`MONGO_URI`), and session secret (`SESSION_SECRET`). The configuration includes fallback values and environment-specific settings.

Middleware Setup Implements key security and application features. It configures cookie parsing, secure session management, and JSON body parsing. It includes CORS headers, request logging, and Passport.js authentication. Session settings are adjusted for development and production environments.

Database Connection Connects to MongoDB using the defined URI. This section includes connection status logging and graceful process termination on failure, with robust error handling.

Route Configuration Sets up all major application routes with specific path prefixes. These include authentication, user management, file uploads, profile, notes, news, weather, coordinates, calendar, friends, and chat. Static file serving is configured for uploads and chat-related files.

WebSocket Initialization Creates an HTTP server and integrates Socket.IO. The WebSocket instance is configured with the Express app and stored using `app.set()` for accessibility across the application.

Server Startup Starts the server with proper port binding and logs startup details. The startup log includes WebSocket endpoint information for easier debugging and monitoring.

3.5 Testing

Page: Login Page

- Test Case 1: Successful manual login
  - Step 1: Open the login page
  - Step 2: Enter valid username and password
  - Step 3: Click "Login" button
  - Expected result: User is redirected to the home page with valid JWT token stored in HTTP-only cookie
- Test Case 2: Login with invalid credentials
  - Step 1: Open the login page
  - Step 2: Enter valid username and incorrect password
  - Step 3: Click "Login" button
  - Expected result: Error message "Wrong username or password" is displayed
- Test Case 3: Login with empty fields
  - Step 1: Open the login page
  - Step 2: Leave username and password fields empty
  - Step 3: Click "Login" button
  - Expected result: Form validation prevents submission
- Test Case 4: Successful Google login
  - Step 1: Open the login page
  - Step 2: Click "Login with Google" button
  - Step 3: Complete Google OAuth flow
  - Expected result: User is redirected to home page with valid JWT token
- Test Case 5: Successful GitHub login
  - Step 1: Open the login page

- Step 2: Click "Login with GitHub" button
- Step 3: Complete GitHub OAuth flow
- Expected result: User is redirected to home page with valid JWT token

Page: Registration Page

- Test Case 1: Successful registration
  - Step 1: Open the registration page
  - Step 2: Fill in all required fields (firstName, lastName, username, email, password)
  - Step 3: Click "Register" button
  - Step 4: Enter verification code from email
  - Expected result: Account is created and user is redirected to home page
- Test Case 2: Registration with existing email
  - Step 1: Open the registration page
  - Step 2: Enter an email that is already registered
  - Step 3: Fill in other required fields
  - Step 4: Click "Register" button
  - Expected result: Error message "Email already exists" is displayed
- Test Case 3: Registration with weak password
  - Step 1: Open the registration page
  - Step 2: Enter a password that doesn't meet requirements
  - Step 3: Fill in other required fields
  - Step 4: Click "Register" button
  - Expected result: Password validation error is displayed

Page: Password Reset Page

- Test Case 1: Successful password reset
  - Step 1: Open the password reset page
  - Step 2: Enter registered email address
  - Step 3: Enter new password
  - Step 4: Enter verification code from email

- Expected result: Password is updated and user can login with new password

- Test Case 2: Reset with non-existent email
  - Step 1: Open the password reset page
  - Step 2: Enter non-existent email address
  - Step 3: Click "Reset" button
  - Expected result: Error message "User not found" is displayed
- Test Case 3: Reset with expired code
  - Step 1: Open the password reset page
  - Step 2: Enter registered email and new password
  - Step 3: Wait for code to expire (15 minutes)
  - Step 4: Enter expired code
  - Expected result: Error message "Verification code has expired" is displayed

Page: Email Verification Page

- Test Case 1: Successful email verification
  - Step 1: Complete registration process
  - Step 2: Open verification code from email
  - Step 3: Enter verification code
  - Expected result: Account is verified and user is redirected to home page
- Test Case 2: Verification with wrong code
  - Step 1: Complete registration process
  - Step 2: Enter incorrect verification code
  - Step 3: Click "Verify" button
  - Expected result: Error message "Invalid verification code" is displayed
- Test Case 3: Verification with expired code
  - Step 1: Complete registration process
  - Step 2: Wait for code to expire (15 minutes)
  - Step 3: Enter expired code

- Expected result: Error message "Verification code has expired" is displayed

Page: Home Page Header Test Case 1: Theme Toggle Functionality

- Step 1: Navigate to home page
- Step 2: Click theme toggle button
- Expected result:
  - Theme changes from light to dark or vice versa
  - Theme preference is saved in localStorage
  - All components update to reflect new theme

Test Case 2: Widget Management

- Step 1: Click "Add Widget" button
- Step 2: Select a widget from the dropdown
- Expected result:
  - Selected widget appears on the dashboard
  - Widget configuration is saved in localStorage
  - Widget is properly positioned in the grid

Test Case 3: Widget Removal

- Step 1: Hover over an existing widget
- Step 2: Click the remove button
- Expected result:
  - Widget is removed from the dashboard
  - Widget configuration is updated in localStorage

Test Case 4: Background Image Management

- Step 1: Click "Backgrounds" button
- Step 2: Click "Upload image" button
- Step 3: Select and upload an image
- Step 4: Click on the image from the list
- Expected result:
  - Background image updates
  - Image is properly scaled and positioned
  - Image preference is saved in localStorage

Test Case 5: Responsive Layout
- Step 1: Resize browser window to mobile size
- Expected result:
  - Burger icon, clock and theme toggle buttons are left
  - Icons and buttons remain accessible from dropdown menu after clicking on burger icon
  - Clock shows only HH:MM format

Test Case 6: User Profile Access
- Step 1: Click user profile icon
- Expected result:
  - Profile dropdown menu appears
  - Shows Settings button
  - Contains logout option

Test Case 7: Logout Functionality
- Step 1: Click user profile icon
- Step 2: Click "Logout" option
- Expected result:
  - User is logged out
  - Redirected to login page
  - Session data cleared

Test Case 8: Session Management
- Step 1: Let session expire
- Step 2: Attempt to perform actions
- Expected result:
  - User is redirected to login
  - Session is properly cleared

Test Case 9: Field Validation

- Step 1: Open settings modal
- Step 2: Click edit button for username field
- Step 3: Enter invalid username (e.g., "a")
- Step 4: Click "Save Changes"
- Expected result: Error snackbar appears with message "Username must be at least 5 lowercase alphanumeric characters"

Test Case 10: Password Change Validation

- Step 1: Click edit button for password fields
- Step 2: Enter different passwords in "New Password" and "Confirm Password"
- Step 3: Click "Change Password"
- Expected result: Error snackbar appears with message "Passwords don't match"

Test Case 11: Avatar Upload

- Step 1: Click on the avatar image
- Step 2: Select an image file
- Step 3: Click "Save Changes"
- Expected result: Avatar updates and success message appears

Test Case 12: Name Fields Validation

- Step 1: Click edit button for First Name
- Step 2: Enter invalid characters (e.g., "John123")
- Step 3: Click "Save Changes"
- Expected result: Error message "First name must contain 3-20 letters only"

Test Case 13: Email Validation

- Step 1: Click edit button for email field
- Step 2: Enter invalid email format
- Step 3: Click "Save Changes"
- Expected result: Error message "Invalid email format"

Test Case 14: Account Deletion

- Step 1: Click "Delete Account" button
- Step 2: Verify confirmation dialog appears
- Step 3: Click "Cancel"
- Step 4: Click "Delete Account" again
- Step 5: Click "Delete" in confirmation
- Expected result: Account is deleted and user is redirected to login page

Test Case 15: Dark Mode Compatibility

- Step 1: Enable dark mode in the application
- Step 2: Open settings modal
- Expected result: Modal appears with dark theme colors and proper contrast

Test Case 16: Field Edit Toggle

- Step 1: Click edit button for any field
- Step 2: Verify field becomes editable
- Step 3: Click edit button again
- Expected result: Field becomes non-editable and retains previous value

Test Case 17: Successful Profile Update

- Step 1: Edit multiple fields (username, first name, last name)
- Step 2: Click "Save Changes"
- Expected result: Success message appears and changes are reflected in the UI

Test Case 18: Modal Close Behavior

- Step 1: Make changes to multiple fields
- Step 2: Click the close (X) button
- Step 3: Reopen the modal
- Expected result: Modal reopens with original values, unsaved changes are discarded

Test Case 19: Password Change Success

- Step 1: Click edit button for password fields
- Step 2: Enter valid matching passwords
- Step 3: Click "Change Password"
- Expected result: Success message appears and password fields are cleared

Test Case 20: Responsive Layout

- Step 1: Open settings on desktop view

- Step 2: Resize browser to mobile width
- Expected result: Layout adjusts responsively, all fields remain accessible

Page: Home Page

- Test Case 1: Initial Layout Loading
  - Step 1: Open the application
  - Step 2: Verify default widgets are loaded ('notes', 'weather', 'calendar', 'news', 'chats')
  - Step 3: Check layout persistence from localStorage
  - Expected result: All default widgets are displayed in their saved positions
- Test Case 2: Theme Toggle
  - Step 1: Click theme toggle button
  - Step 2: Verify theme changes
  - Step 3: Refresh page
  - Expected result: Theme persists after refresh and affects all components
- Test Case 3: Widget Addition
  - Step 1: Remove a widget
  - Step 2: Click "Add Widget" button
  - Step 3: Select removed widget from dropdown
  - Expected result: Widget appears in default position with correct dimensions
- Test Case 4: Widget Removal
  - Step 1: Hover over a widget
  - Step 2: Click remove button
  - Step 3: Check localStorage
  - Expected result: Widget disappears, layout updates, changes persist in localStorage
- Test Case 5: Background Image
  - Step 1: Change background image

- Step 2: Refresh page
- Step 3: Verify image persistence
- Expected result: Custom background displays and persists after refresh

Page: Desk Component

- Test Case 6: Grid Layout Responsiveness
  - Step 1: Open on desktop view (>768px)
  - Step 2: Verify 8-column layout
  - Step 3: Resize to tablet view (<768px)
  - Step 4: Verify 4-column layout
  - Step 5: Resize to mobile view (<640px)
  - Step 6: Verify 2-column layout
  - Expected result: Layout adjusts properly at each breakpoint
- Test Case 7: Widget Dragging (Desktop)
  - Step 1: Attempt to drag widget on desktop
  - Step 2: Release at new position
  - Step 3: Verify position saves
  - Expected result: Widget moves and maintains new position after refresh
- Test Case 8: Widget Resizing (Desktop)
  - Step 1: Grab resize handle of widget
  - Step 2: Resize within min/max constraints
  - Step 3: Verify size persists
  - Expected result: Widget resizes within constraints and maintains size
- Test Case 9: Mobile Layout Restrictions
  - Step 1: Access on mobile view
  - Step 2: Attempt to drag widget
  - Step 3: Attempt to resize widget
  - Expected result: Dragging and resizing are disabled on mobile
- Test Case 10: Geolocation Services
  - Step 1: Load page with weather widget

  - Step 2: Accept location permissions
  - Step 3: Verify weather data loads
  - Expected result: Weather widget shows local weather data
- Test Case 11: Fallback Location
  - Step 1: Deny location permissions
  - Step 2: Check weather widget
  - Expected result: Weather widget shows default location (London) data
- Test Case 12: Layout Collision Prevention
  - Step 1: Drag widget over another widget
  - Step 2: Attempt to overlap widgets
  - Expected result: Widgets cannot overlap due to collision prevention
- Test Case 13: Single Widget Mobile Mode
  - Step 1: Load page on mobile device
  - Step 2: Verify only one widget displays
  - Step 3: Switch to different widget
  - Expected result: Only one widget shows at a time on mobile
- Test Case 14: Layout State Persistence
  - Step 1: Arrange widgets in custom layout
  - Step 2: Close browser
  - Step 3: Reopen application
  - Expected result: Layout restores to last saved state
- Test Case 15: Error Handling
  - Step 1: Simulate API failure
  - Step 2: Check error snackbar
  - Step 3: Verify fallback behavior
  - Expected result: Error message displays and fallback content shows

Widget: Notes - Test Case 1: Note Creation

- Step 1: Click "Add Note" button

- Step 2: Enter note content

Expected result:

- New note appears in the list
- Note has correct title(beginning of content) and content
- Note persists after page refresh

Test Case 2: Note Editing
- Step 1: Clickon existing note
- Step 2: Modify content
- Step 3: Refresh page

Expected result:

- Note updates with new content
- Changes persist after refresh

Test Case 3: Note Deletion
- Step 1: Click delete icon on note

Expected result:

- Note is removed from list
- Deletion persists after refresh

Test Case 4: Note Color Change
- Step 1: Click color toggle icon in header
- Step 2: Refresh page

Expected result:

- Note background changes to selected color
- Color change persists after refresh

Test Case 5: Multiple Note Operations
- Step 1: Select different note type

Expected result: Note automatically changes it's type

Test Case 6: Responsive Design
- Step 1: View on desktop
- Step 2: Resize to tablet
- Step 3: Resize to mobile

Expected result:

- Layout adapts to screen size
- All features remain accessible
- Note grid adjusts columns appropriately

Widget: Calendar

- Test Case 1: Event Creation
  - Step 1: Click on a calendar date
  - Step 2: Fill in event details in EventModal:
    - Title: "Team Meeting"
    - Description: "Weekly sync"
    - Start time: "10:00"
    - End time: "11:00"
  - Step 3: Click "Save" button
  - Expected result:
    - New event appears on calendar
    - Event displays correct time slot
    - Success notification appears
- Test Case 2: Event Time Validation
  - Step 1: Open event creation modal
  - Step 2: Set end time earlier than start time
  - Step 3: Attempt to save
  - Expected result:
    - Validation error appears
    - Save button remains disabled
    - Error message indicates invalid time range
- Test Case 3: Event Editing
  - Step 1: Click existing event
  - Step 2: In EditModal, modify:
    - Change title
    - Update time
    - Edit description

  - Step 3: Save changes
  - Expected result:
    * Event updates with new details
    * Position adjusts if time changed
    * Changes persist after refresh

- Test Case 4: View Switching
  - Step 1: Test view switches:
    * Month view
    * Week view
    * Day view
  - Step 2: Navigate between periods
  - Expected result:
    * View changes correctly
    * Events display appropriately in each view
    * Navigation maintains event visibility

- Test Case 5: Event Deletion
  - Step 1: Open existing event
  - Step 2: Click delete button
  - Step 3: Confirm deletion
  - Expected result:
    * Event removes from calendar
    * Success notification appears
    * Deletion persists after refresh

- Test Case 6: Recurring Events
  - Step 1: Create event with recurrence
  - Step 2: Set weekly repeat
  - Step 3: Check multiple weeks
  - Expected result:
    * Event appears on specified days
    * Recurrence pattern is correct

        * All instances are editable

- Test Case 7: Date Navigation
  - Step 1: Use navigation arrows
  - Step 2: Use today button
  - Step 3: Use date picker
  - Expected result:
    * Calendar navigates correctly
    * Events load for new dates
    * Today highlights properly
- Test Case 8: Event Overlap Handling
  - Step 1: Create overlapping events
  - Step 2: View in different calendar views
  - Expected result:
    * Events display without visual conflicts
    * Time slots show multiple events clearly
    * All events remain accessible
- Test Case 9: Responsive Layout
  - Step 1: View on desktop
  - Step 2: Resize to tablet
  - Step 3: View on mobile
  - Expected result:
    * Layout adapts to screen size
    * Events remain readable
    * Modals are properly sized
- Test Case 10: Theme Compatibility
  - Step 1: Test in light mode:
    * Create event
    * View different calendar views
  - Step 2: Switch to dark mode
  - Expected result:

        * Calendar adapts to theme
        * Events maintain visibility
        * Modals follow theme

Widget: News

- Test Case 1: Initial News Loading
  - Step 1: Load news widget
  - Step 2: Verify loading state appears (CircularProgress)
  - Step 3: Wait for API response
  - Expected result:
    * Loading indicator shows and then disappears
    * Article appears with title, description, source name
    * Publication date is correctly formatted
    * Content adapts to current theme (dark/light)
- Test Case 2: Article Auto-Rotation
  - Step 1: Load news widget with initial article
  - Step 2: Wait for 5 minutes (300000ms)
  - Step 3: Observe article change
  - Expected result:
    * New random article appears automatically
    * Transition is smooth
    * Article details update correctly
    * Previous article is different from new one
- Test Case 3: Location-based News
  - Step 1: Load widget with specific countryCode
  - Step 2: Change countryCode prop
  - Step 3: Verify news update
  - Expected result:
    * News articles fetch for new location
    * Loading state shows during update
    * Articles are relevant to new location
    * Error handling if location is invalid

- Test Case 4: Article Interaction
  - Step 1: Click "See more.." link
  - Step 2: Verify new tab opens
  - Step 3: Test click propagation
  - Expected result:
    - Article opens in new tab
    - Original tab remains unchanged
    - URL has noopener and noreferrer attributes
    - Click doesn't affect widget container
- Test Case 5: Responsive Layout
  - Step 1: View widget at width $>$ 400px
  - Step 2: Resize to width $<$ 400px
  - Step 3: Verify layout changes
  - Expected result:
    - Layout switches from row to column
    - Typography adjusts (subtitle1 to subtitle2)
    - Content remains readable
    - Source name and date remain visible

Page: Home Page

- Test Case 1: Initial Layout Loading
  - Step 1: Open the application
  - Step 2: Verify default widgets are loaded ('notes', 'weather', 'calendar', 'news', 'chats')
  - Step 3: Check layout persistence from localStorage
  - Expected result: All default widgets are displayed in their saved positions
- Test Case 2: Theme Toggle
  - Step 1: Click theme toggle button
  - Step 2: Verify theme changes
  - Step 3: Refresh page

  - Expected result: Theme persists after refresh and affects all components

- Test Case 3: Widget Addition
  - Step 1: Remove a widget
  - Step 2: Click "Add Widget" button
  - Step 3: Select removed widget from dropdown
  - Expected result: Widget appears in default position with correct dimensions
- Test Case 4: Widget Removal
  - Step 1: Hover over a widget
  - Step 2: Click remove button
  - Step 3: Check localStorage
  - Expected result: Widget disappears, layout updates, changes persist in localStorage
- Test Case 5: Background Image
  - Step 1: Change background image
  - Step 2: Refresh page
  - Step 3: Verify image persistence
  - Expected result: Custom background displays and persists after refresh

Page: Desk Component

- Test Case 6: Grid Layout Responsiveness
  - Step 1: Open on desktop view (>768px)
  - Step 2: Verify 8-column layout
  - Step 3: Resize to tablet view (<768px)
  - Step 4: Verify 4-column layout
  - Step 5: Resize to mobile view (<640px)
  - Step 6: Verify 2-column layout
  - Expected result: Layout adjusts properly at each breakpoint
- Test Case 7: Widget Dragging (Desktop)
  - Step 1: Attempt to drag widget on desktop
  - Step 2: Release at new position

  - Step 3: Verify position saves
  - Expected result: Widget moves and maintains new position after refresh

- Test Case 8: Widget Resizing (Desktop)
  - Step 1: Grab resize handle of widget
  - Step 2: Resize within min/max constraints
  - Step 3: Verify size persists
  - Expected result: Widget resizes within constraints and maintains size

- Test Case 9: Mobile Layout Restrictions
  - Step 1: Access on mobile view
  - Step 2: Attempt to drag widget
  - Step 3: Attempt to resize widget
  - Expected result: Dragging and resizing are disabled on mobile

- Test Case 10: Geolocation Services
  - Step 1: Load page with weather widget
  - Step 2: Accept location permissions
  - Step 3: Verify weather data loads
  - Expected result: Weather widget shows local weather data

- Test Case 11: Fallback Location
  - Step 1: Deny location permissions
  - Step 2: Check weather widget
  - Expected result: Weather widget shows default location (London) data

- Test Case 12: Layout Collision Prevention
  - Step 1: Drag widget over another widget
  - Step 2: Attempt to overlap widgets
  - Expected result: Widgets cannot overlap due to collision prevention

- Test Case 13: Single Widget Mobile Mode
  - Step 1: Load page on mobile device
  - Step 2: Verify only one widget displays

  - Step 3: Switch to different widget
  - Expected result: Only one widget shows at a time on mobile
- Test Case 14: Layout State Persistence
  - Step 1: Arrange widgets in custom layout
  - Step 2: Close browser
  - Step 3: Reopen application
  - Expected result: Layout restores to last saved state
- Test Case 15: Error Handling
  - Step 1: Simulate API failure
  - Step 2: Check error snackbar
  - Step 3: Verify fallback behavior
  - Expected result: Error message displays and fallback content shows

Widget: Notes

- Test Case 1: Note Creation
  - Step 1: Click "Add Note" button
  - Step 2: Enter note content
  - Step 3: Click save
  - Expected result: New note appears in the list and persists after refresh
- Test Case 2: Note Editing
  - Step 1: Click edit icon on existing note
  - Step 2: Modify content
  - Step 3: Save changes
  - Expected result: Note updates with new content
- Test Case 3: Note Deletion
  - Step 1: Click delete icon on note
  - Step 2: Confirm deletion
  - Expected result: Note is removed from list

Widget: Weather

- Test Case 1: Location Detection
  - Step 1: Load weather widget
  - Step 2: Allow location access

  - Expected result: Weather data for current location displays
- Test Case 2: Manual Location Search
  - Step 1: Click location search field
  - Step 2: Enter city name
  - Step 3: Select from dropdown
  - Expected result: Weather updates for selected location
- Test Case 3: Forecast View
  - Step 1: Click forecast tab
  - Step 2: Check multiple day forecast
  - Expected result: Shows 5-day weather forecast

Widget: Calendar

- Test Case 1: Event Creation
  - Step 1: Click on a date
  - Step 2: Fill event details
  - Step 3: Save event
  - Expected result: Event appears on calendar
- Test Case 2: Event Editing
  - Step 1: Click existing event
  - Step 2: Modify details
  - Step 3: Save changes
  - Expected result: Event updates with new details
- Test Case 3: View Switching
  - Step 1: Switch between month/week/day views
  - Step 2: Navigate between periods
  - Expected result: Calendar displays correct view and events

Widget: News

- Test Case 1: News Loading
  - Step 1: Open news widget
  - Step 2: Verify news articles load
  - Expected result: Current news articles display with titles and descriptions

- Test Case 2: Article Rotation
  - Step 1: Wait for article rotation interval
  - Step 2: Verify new article appears
  - Expected result: Articles rotate automatically every 5 minutes
- Test Case 3: Article Interaction
  - Step 1: Click on article
  - Step 2: Verify link opens
  - Expected result: Article opens in new tab

Widget: Chats

- Test Case 1: Friend Search
  - Step 1: Click search field
  - Step 2: Enter username
  - Step 3: Send friend request
  - Expected result: Friend request sent successfully
- Test Case 2: Message Sending
  - Step 1: Select chat conversation
  - Step 2: Type message
  - Step 3: Send message
  - Expected result: Message appears in chat with correct status
- Test Case 3: File Attachment
  - Step 1: Click attachment button
  - Step 2: Select file
  - Step 3: Send message with attachment
  - Expected result: File uploads and appears in chat
- Test Case 4: Real-time Updates
  - Step 1: Open chat in two browsers
  - Step 2: Send message from one
  - Step 3: Check real-time delivery
  - Expected result: Message appears instantly in other browser
- Test Case 5: Friend Request Management

  - Step 1: Open friend requests
  - Step 2: Accept/Reject request
  - Expected result: Friend list updates accordingly

Common Widget Tests

- Test Case 1: Widget Resizing
  - Step 1: Grab resize handle
  - Step 2: Resize widget
  - Step 3: Check content adaptation
  - Expected result: Content adjusts to new size properly
- Test Case 2: Error States
  - Step 1: Simulate network error
  - Step 2: Check error display
  - Step 3: Verify retry functionality
  - Expected result: Error message shows with retry option
- Test Case 3: Loading States
  - Step 1: Trigger data reload
  - Step 2: Verify loading indicator
  - Expected result: Loading state displays properly
- Test Case 4: Theme Compatibility
  - Step 1: Switch theme
  - Step 2: Check widget appearance
  - Expected result: Widget adapts to current theme
- Test Case 5: Mobile Responsiveness
  - Step 1: View on mobile device
  - Step 2: Check functionality
  - Step 3: Verify usability
  - Expected result: Widget is fully functional on mobile

Widget: Notes

- Test Case 1: Note Creation
  - Step 1: Click "Add Note" button
  - Step 2: Enter note title

  - Step 3: Enter note content
  - Step 4: Click save button
  - Expected result:
    * New note appears in the list
    * Note has correct title and content
    * Success notification appears
    * Note persists after page refresh
- Test Case 2: Note Validation
  - Step 1: Click "Add Note" button
  - Step 2: Leave title empty
  - Step 3: Click save button
  - Expected result: Validation error message appears
- Test Case 3: Note Editing
  - Step 1: Click edit icon on existing note
  - Step 2: Modify title and content
  - Step 3: Click save button
  - Step 4: Refresh page
  - Expected result:
    * Note updates with new content
    * Success notification appears
    * Changes persist after refresh
- Test Case 4: Note Deletion
  - Step 1: Click delete icon on note
  - Step 2: Confirm deletion in dialog
  - Step 3: Refresh page
  - Expected result:
    * Note is removed from list
    * Success notification appears
    * Deletion persists after refresh
- Test Case 5: Note Search

  - Step 1: Create multiple notes
  - Step 2: Enter search term in search field
  - Step 3: Clear search field
  - Expected result:
    * Only matching notes display
    * All notes reappear when search is cleared

- Test Case 6: Note Color Change
  - Step 1: Click color palette icon on note
  - Step 2: Select different color
  - Step 3: Refresh page
  - Expected result:
    * Note background changes to selected color
    * Color change persists after refresh

- Test Case 7: Note Sorting
  - Step 1: Create notes with different dates
  - Step 2: Click sort button
  - Step 3: Toggle between ascending/descending
  - Expected result: Notes reorder based on selected sort direction

- Test Case 8: Offline Functionality
  - Step 1: Disable internet connection
  - Step 2: Attempt to create/edit/delete note
  - Step 3: Restore connection
  - Expected result:
    * Error message appears during offline actions
    * Changes sync when connection restores

- Test Case 9: Multiple Note Operations
  - Step 1: Select multiple notes using checkboxes
  - Step 2: Apply bulk action (delete/color change)
  - Step 3: Confirm action
  - Expected result: Action applies to all selected notes

- Test Case 10: Note Input Component
  - Step 1: Open note input
  - Step 2: Test auto-resize functionality
  - Step 3: Test character limit
  - Expected result:
    - Input resizes with content
    - Cannot exceed character limit
- Test Case 11: Error Handling
  - Step 1: Simulate API error
  - Step 2: Attempt note operation
  - Expected result:
    - Error notification appears
    - UI remains responsive
    - Data integrity maintained
- Test Case 12: Loading States
  - Step 1: Load notes widget
  - Step 2: Perform note operation
  - Expected result:
    - Loading indicator shows during operations
    - UI remains responsive
- Test Case 13: Responsive Design
  - Step 1: View on desktop
  - Step 2: Resize to tablet
  - Step 3: Resize to mobile
  - Expected result:
    - Layout adapts to screen size
    - All features remain accessible
    - Note grid adjusts columns appropriately
- Test Case 14: Theme Compatibility
  - Step 1: Switch to dark mode

  - Step 2: Create and edit notes
  - Step 3: Switch to light mode
  - Expected result:
    * Notes are readable in both themes
    * Color contrasts maintain accessibility
    * UI elements adapt to theme

- Test Case 15: API Integration
  - Step 1: Monitor network calls during operations
  - Step 2: Verify request/response format
  - Step 3: Check error handling
  - Expected result:
    * API calls are properly formatted
    * Responses are handled correctly
    * Error states are managed appropriately

Widget: Calendar

- Test Case 1: Event Creation
  - Step 1: Click on a calendar date
  - Step 2: Fill in event details in EventModal:
    * Title: "Team Meeting"
    * Description: "Weekly sync"
    * Start time: "10:00"
    * End time: "11:00"
  - Step 3: Click "Save" button
  - Expected result:
    * New event appears on calendar
    * Event displays correct time slot
    * Success notification appears

- Test Case 2: Event Time Validation
  - Step 1: Open event creation modal
  - Step 2: Set end time earlier than start time
  - Step 3: Attempt to save

  - Expected result:
    - Validation error appears
    - Save button remains disabled
    - Error message indicates invalid time range

- Test Case 3: Event Editing
  - Step 1: Click existing event
  - Step 2: In EditModal, modify:
    - Change title
    - Update time
    - Edit description
  - Step 3: Save changes
  - Expected result:
    - Event updates with new details
    - Position adjusts if time changed
    - Changes persist after refresh

- Test Case 4: View Switching
  - Step 1: Test view switches:
    - Month view
    - Week view
    - Day view
  - Step 2: Navigate between periods
  - Expected result:
    - View changes correctly
    - Events display appropriately in each view
    - Navigation maintains event visibility

- Test Case 5: Event Deletion
  - Step 1: Select existing event
  - Step 2: Click delete button
  - Expected result:
    - Event removes from calendar

        * Success notification appears
        * Deletion persists after refresh
- Test Case 6: Date Navigation
  - Step 1: Use navigation arrows
  - Step 2: Use today button
  - Step 3: Use date picker
  - Expected result:
    * Calendar navigates correctly
    * Events load for new dates
    * Today highlights properly

Widget: News

- Test Case 1: Initial News Loading
  - Step 1: Load news widget
  - Step 2: Verify loading state appears (CircularProgress)
  - Step 3: Wait for API response
  - Expected result:
    * Loading indicator shows and then disappears
    * Article appears with title, description, source name
    * Publication date is correctly formatted
    * Content adapts to current theme (dark/light)
- Test Case 2: Article Auto-Rotation
  - Step 1: Load news widget with initial article
  - Step 2: Wait for 5 minutes (300000ms)
  - Step 3: Observe article change
  - Expected result:
    * New random article appears automatically
    * Article details update correctly
    * Previous article is different from new one
- Test Case 3: Location-based News
  - Step 1: Change location in Weather widget

  - Step 2: Verify news update
  - Expected result:
    * News articles fetch for new location
    * Loading state shows during update
    * Articles are relevant to new location
    * Error handling if location is invalid

- Test Case 4: Article Interaction
  - Step 1: Click "See more.." link
  - Step 2: Verify new tab opens
  - Step 3: Test click propagation
  - Expected result:
    * Article opens in new tab
    * Original tab remains unchanged
    * URL has noopener and noreferrer attributes
    * Click doesn't affect widget container

Widget: Weather

- Test Case 1: Initial Weather Loading
  - Step 1: Load weather widget with valid city
  - Step 2: Verify loading state (WidgetLoading component)
  - Step 3: Wait for API response
  - Expected result:
    * Loading indicator shows and disappears
    * Current temperature displays
    * Weather condition icon appears
    * City name is visible

- Test Case 2: City Name Change
  - Step 1: Click edit icon next to city name
  - Step 2: Enter new city name in dialog
  - Step 3: Click "Confirm" button
  - Expected result:
    * Dialog opens

        * New city name updates after confirmation
        * Dialog closes after confirmation
        * Weather data updates for new location
- Test Case 3: Responsive Layout - Small Width
    - Step 1: Resize window to width $<$ 300px
    - Step 2: Observe layout changes
    - Expected result:
        * Layout switches to vertical orientation
        * Font sizes adjust (h5 to h6)
        * Weather icon size reduces (24px to 16px)
        * Padding adjusts for smaller screen
- Test Case 4: Responsive Layout - Short Height
    - Step 1: Resize window to height $<$ 200px
    - Step 2: Observe layout changes
    - Expected result:
        * "Feels like" temperature hides
        * Hourly forecast section hides
        * Maintains essential information visibility
        * Layout remains functional
- Test Case 5: Theme Switching
    - Step 1: Load widget in light mode
    - Step 2: Switch to dark mode
    - Step 3: Verify theme changes
    - Expected result:
        * Background color changes
        * Text color adjusts
        * Weather icons invert colors appropriately
        * Dialog theme updates
- Test Case 6: Hourly Forecast Display
    - Step 1: Load widget with sufficient size

  - Step 2: Verify next 5 hours forecast
  - Expected result:
    * Shows exactly 5 future hours
    * Each hour shows correct time
    * Each hour displays temperature
    * Each hour shows weather condition icon

Widget: Chats

- Test Case 1: Initial Chat Loading
  - Step 1: Load chat widget
  - Step 2: Verify initial state
  - Expected result:
    * Friends list displays correctly
    * "Select a friend to start chatting" message shows
    * Default avatar placeholder visible
    * Theme matches system settings
- Test Case 2: Friend Search
  - Step 1: Enter text in friend search field
  - Step 2: Wait for search results
  - Expected result:
    * Search results appear after 500ms debounce
    * Results filter correctly
    * Friend list updates dynamically
    * Empty state shows if no matches
- Test Case 3: Add Friend Flow
  - Step 1: Click "Add Friend" button
  - Step 2: Search for user
  - Step 3: Send friend request
  - Expected result:
    * Dialog opens with search field
    * User results display with avatars
    * Request sends successfully

        * Toast notification appears

- Test Case 4: Friend Request Management
  - Step 1: Receive friend request
  - Step 2: Accept/Reject request
  - Expected result:
    * Request appears in pending list
    * Badge count updates
    * Appropriate toast shows on action
    * Friend list updates on accept
- Test Case 5: Message Sending
  - Step 1: Select friend
  - Step 2: Type message
  - Step 3: Send message
  - Expected result:
    * Message appears in chat
    * Scrolls to new message
    * Status indicator shows
    * Input field clears
- Test Case 6: File Attachment
  - Step 1: Click attachment button
  - Step 2: Select multiple files
  - Step 3: Send with attachments
  - Expected result:
    * File preview shows
    * Size limits enforce
    * Upload progress indicates
    * Files send successfully
- Test Case 7: Message Status Updates
  - Step 1: Send message
  - Step 2: Recipient reads message

  - Expected result:
    - Status changes from sent to read
    - Real-time update occurs
    - Socket event processes
    - UI reflects new status

- Test Case 8: Typing Indicators
  - Step 1: Start typing message
  - Step 2: Stop typing
  - Expected result:
    - "username is typing..." appears
    - Indicator shows in real-time
    - Disappears after stopping
    - Socket events trigger correctly

- Test Case 9: Responsive Layout
  - Step 1: View at width > 450px
  - Step 2: Resize to width <= 450px
  - Expected result:
    - Layout switches to mobile view
    - Menu button appears
    - Friend list moves to dropdown
    - Chat area adjusts properly

- Test Case 10: User Status Management
  - Step 1: Change tab visibility
  - Step 2: Observe friend status
  - Expected result:
    - Status updates to away/online
    - Friend list reflects changes
    - Status dot color changes
    - Real-time updates occur

- Test Case 11: Message History Loading

  - Step 1: Select different friends
  - Step 2: Switch between chats
  - Expected result:
    * Messages load correctly
    * Chat history preserves
    * Scroll position maintains
    * Unread counts update

- Test Case 12: Friend Removal
  - Step 1: Click remove friend button
  - Step 2: Confirm removal
  - Expected result:
    * Friend removes from list
    * Chat history clears
    * Socket connection updates
    * Success notification shows

- Test Case 13: Theme Switching
  - Step 1: Switch between dark/light modes
  - Step 2: Verify UI elements
  - Expected result:
    * Colors update appropriately
    * Icons adapt to theme
    * Text remains readable
    * Modals match theme

- Test Case 14: Unread Message Handling
  - Step 1: Receive new message
  - Step 2: Check unread indicators
  - Expected result:
    * Badge shows unread count
    * Count updates in real-time
    * Clears on reading messages

        * Updates across friend list

- Test Case 15: Error Handling
  - Step 1: Trigger various errors
  - Step 2: Observe error handling
  - Expected result:
    * Socket reconnection works
    * Error messages display

Chapter 4

Conclusion

This application was created to showcase the skills acquired during the pursuit of a Computer Science degree at ELTE University. The documentation thoroughly describes all aspects of the application from the developer's and user's perspectives. The project can be expanded with new functionality but is already ready for full use by users. **Your Personal Space** will undoubtedly enhance the productivity of people engaged in intellectual work, and it can also be an excellent opportunity for developers to improve their skills by contributing to the enhancement of this application.

In addition to addressing important tasks for users, the application also opens up unique opportunities for those who wish to improve their skills in the field of application development and design. Developers involved in the further development of the project can expand its functionality, improve the user interface, or introduce innovative ideas, which contributes to their professional growth.

It is important to note that **Your Personal Space** is not limited to individual users. The project always has room for new ideas and partners. Anyone can contribute, and participation in the project opens up new horizons for learning and professional development. This interaction allows for the exchange of experiences, learning from others, and at the same time creating a high-quality product.

Bibliography

[1] React — A JavaScript library for building user interfaces.
https://react.dev/

[2] Tailwind CSS — A utility-first CSS framework.
https://tailwindcss.com/docs

[3] MongoDB — The developer data platform.
https://www.mongodb.com/docs/

[4] Mongoose — Elegant MongoDB object modeling for Node.js.
https://mongoosejs.com/docs/

[5] Material-UI (MUI) — React components for faster web development.
https://mui.com/material-ui/getting-started/overview/

[6] Socket.IO — Real-time bidirectional event-based communication.
https://socket.io/docs/

[7] uuid — RFC-compliant UUID generation for JavaScript.
https://github.com/uuidjs/uuid

[8] date-fns — Modern JavaScript date utility library.
https://date-fns.org/docs/Getting-Started

[9] React Toastify — Toast notifications for React.
https://fkhadra.github.io/react-toastify/introduction/

[10] Express.js — Fast, unopinionated, minimalist web framework for Node.js.
https://expressjs.com/en/starter/installing.html

[11] jsonwebtoken — Implementation of JSON Web Tokens.
https://github.com/auth0/node-jsonwebtoken

[12] bcrypt — Library to hash passwords securely.
https://github.com/kelektiv/node.bcrypt.js

[13] Passport.js — Simple, unobtrusive authentication for Node.js.
http://www.passportjs.org/docs/

[14] Multer — Middleware for handling ‘multipart/form-data‘.
https://github.com/expressjs/multer

[15] Nodemailer — Easy as cake email sending from Node.js.
https://nodemailer.com/about/

[16] OpenCage Geocoding API — Forward and reverse geocoding service.
https://opencagedata.com/api

[17] NewsData.io API — News data and headlines API.
https://newsdata.io/docs

[18] Visual Crossing Weather API — Global weather data and forecasts.
https://www.visualcrossing.com/weather-api

List of Tables

List of Codes