# 1 Motivation and Purpose

## 1.1 Development Motivation

Due to factors such as a fast-paced lifestyle, some individuals who require scheduled medication find themselves neglect to save medication instructions and end up missing dosages or forgetting the correct dosage. This application aims to create a centralized platform focused on medications to help people address these issues and also facilitate individuals interested in drugs to get knowledge about them.

## 1.2 Application Features

The Medication Information Database stores medication instructions, allowing users to retrieve information through searches. The Medication Management System enables users to set up medication plans, while the temporary storage and login functionality preserve user records. User-friendly interface features prioritize a design that is concise and clear, enhancing feedback and accessibility to improve user experience.

## 1.3 Inspirations

Many national drug administrations (like FDA) have medication databases, but few offer APIs with detailed instructions. Existing to-do list apps lack dedicated templates for medication management. We aim to integrate and enhance these features.

# 2 Design and Development Planning

## 2.1 Plan the Development

**Step 1:** The first step is to assign tasks. Our team consists of four members, with one primarily focusing on UI design, and three mainly responsible for coding. I am one of those who is responsible for coding.

**Step 2:** The second step is UI design. The team member responsible for UI design creates the UI and shares it periodically to gather feedback from other team members.

**Step 3:** Third step is iterative development. Continuously update deliverables and conduct testing. In this process, I am primarily responsible for coding the homepage and login interface. Upon request from other team members, I frequently provide assistance and address errors and failures for them.

## 2.2 Prototype Designed

After confirming the initial low-fidelity prototype, we began designing the high-fidelity pages of the app. We focused on creating detailed design specifications, covering colours, fonts, and components, ensuring a positive and reliable brand tone.

Users log in, enter the app's homepage, search for "cold" to view medications, select one, add it to their schedule with reminders for time, quantity, and frequency. Reminder settings are stored in the medication calendar. Users can manage reminders by clicking "Manage" in the calendar interface.

Later, during the code implementation, we took into consideration that some users may prefer not to log in. Therefore, we navigated the root path to the Home page and maintained a button to navigate to the Login page.

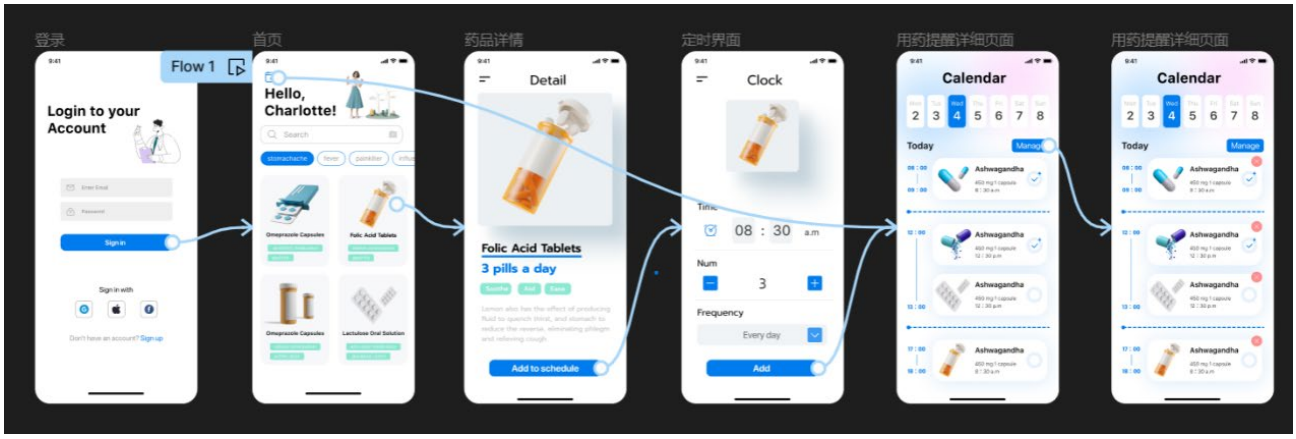


Figure 1 UI and User Flow

# 3 Implementation Details

In this team project, I am mainly responsible for compiling the “Login” and “Home” interface. Meanwhile, due to the poor programming foundation of other team members, I responded to their requests to remedy the “Clock” and “Calendar” interface they were responsible for.

## 3.1 Implementations in My Work

In this part, I will provide detailed documentation on the Login and Home sections that I was responsible for, as well as the implementations I made while modifying the Clock and Calendar sections.

### 3.1.1 Responsive Design

During programming of Login and Home page to which I was responsible, I made an attempt on the responsive design. The relevant effects will be showcased in *Figure 2* and *Figure 3* of the *4.1Simple Acceptance Testing*.

#### 3.1.1.1 Media Queries

Employing CSS3 media queries to apply different style rules depending on device characteristics, enabling adaptation to diverse screens and devices. The following code will set a special style for screen widths greater than 600px.

```
.blockbar{
  width: 100%;
  height: 10vw;
  margin: 0 auto;
  display: block;
  border-radius: 8px;
}
.reminder{
  font-size: 4vw;
  padding: 2vw;
}
```

```
@media screen and(min-width:600px){
  .blockbar{
    width: 100%;
    height: 7vh;
    margin: 0 auto;
    display: block;
    border-radius: 8px;
  }
  .reminder{
    font-size: 4vh;
    padding: 1vh;
  }
}
```

### 3.1.1.2 Grid Layout

Grid layout allows the division of a page into multiple areas and arranges elements flexibly, enabling them to adapt and rearrange based on screen size and device characteristics. The code automatically adjusts the number of elements per row based on screen width (>600px) and fixes it to 2 elements when width is <=600px.

```
.DrugLinks {
  display: grid;
  grid-template-columns:
repeat(auto-fill, minmax(32%, 32vw));
  column-gap: 6vw;
  row-gap: 5vw;
  margin-top: 5vw;
}
```

```
@media screen and (min-width: 600px) {
  /*Other CSS styles*/
  .DrugLinks {
    display: grid;
    grid-template-columns:
repeat(auto-fill, minmax(126px,
126px));
    column-gap: 0px;
    justify-content: space-between;
    row-gap: 5vw;
    margin-top: 5vw;
  }
}
```

### 3.1.1.3 Fluid Layouts

Adopting fluid layouts that dynamically adjust elements based on viewport size rather than fixed layouts. Incorporating adaptable images and media elements to ensure proper display across various devices.

```
.maincontainer {
  width: 70%;
  margin: 0 auto;
}
.titles{
  position: absolute;
  top: 10%;
  font-size: 2vh;
  display: inline-block;
  vertical-align:super;
  max-width: 70%;
}
```

```
.signinWith{
  text-align: center;
  margin-top: 10vw;
}
.decoImg {
  position: absolute;
  bottom: 0;
  right: 0;
  height: 15vh;
  width: auto;
  z-index: -1;
}
```

## 3.1.2 CSS Modules

When integrating CSS files from other team members, we found that CSS namespace pollution occurred. I have searched for a solution and decided to use CSS Modules to prevent interference of styles with the same name. The following is the way to use CSS Modules in React.

Firstly, release the packaged configuration file for React.

```
npm run eject
```

After executing the above command, navigate to the newly appeared “config/” directory and open “webpack.config.js”. Locate the following two rules which match regular CSS and modular CSS files with “cssRegex” and “cssModuleRegex”. Change the “mode” property for both rules to “local”.

```
{
  test: cssRegex,
  exclude: cssModuleRegex,
  use: getStyleLoaders({
    importLoaders: 1,
    sourceMap: isEnvProduction
      ? shouldUseSourceMap
      : isEnvDevelopment,
    modules: {
      mode: 'local',
    },
  }),
  sideEffects: true,
},
```

```
{
  test: cssModuleRegex,
  use: getStyleLoaders({
    importLoaders: 1,
    sourceMap: isEnvProduction
      ? shouldUseSourceMap
      : isEnvDevelopment,
    modules: {
      mode: 'local',
      getLocalIdent:
getCSSModuleLocalIdent,
    },
  }),
},
```

Renaming the ".css" file to the ".module.css" file.

`# DrugLinks.css` to `# DrugLinks.module.css`

The format for referencing a single className and referencing multiple classNames is as follows:

```
className={styles.decoImg}
className={`${styles.blockbar} ${styles.input}`}
```

## 3.1.3 localStorage Maintenance

I have utilized localStorage in this project to maintain account information and calendar items due to the limitations of this.state in maintaining semi-permanent or global variables. localStorage is part of the Web Storage API that allows developers to store key-value pair data in the user's browser. It provides a simple way to permanently store data on the user's browser, even if the user closes the browser or restarts the computer.

For example, use getItem() method to get data named “calendarItems”. As long as localStorage can only store data in String type, JOSN.parse() is needed. After that, check if the data is null and initialize it as an array while it is to prevent error occur at following push() method. At last, use setItem() and JSON.stringify() to update the data in String type after new item is added.

```
    // Retrieve stored strings from localStorage
    const storedItems = localStorage.getItem('calendarItems');
    let items = JSON.parse(storedItems);
    // Parse the stored string into an array, if not, create an empty array
    items = items ? items : [];
    // Add newItem to the array the convert the array to a string and store it back
    items.push(newItem);
    localStorage.setItem('calendarItems', JSON.stringify(items));
```

## 3.1.4 State Maintenance

There are many places where State Maintenance is used, typically divided into three steps: First, use “props” and “this.state” in the constructor to maintain state variables. Then, create methods where “setState()” is called to update state variables. Finally, in the “render()” method, call the methods that can update state variables when the content in the input bar changes, a button is pressed, or when calling an API to fetch data etc.

For example, in the code implementation of the login interface, I maintain “username” and “password” for later simulating user authentication. Of course, there are many other state variables and their implementation logic, which I have hidden in the following code for your convenience of check.

```
        this.state = {
                //Other state variables
                username: "", password: "",
        };
```

Create functions to handle the updating of username and password state by using setState().

```
handleUsernameChange=(e)=> {this.setState({ username: e.target.value });};
handlePasswordChange=(e)=> {this.setState({ password: e.target.value });};
```

Call functions to transmit the changes in input bar.

```
<input {/* other codes */} onChange={this.handleUsernameChange}/>
<input {/* other codes */} onChange={this.handlePasswordChange}/>
```

## 3.1.5 API Operations

I implemented both instances of API calls in this project. The first one involves storing predefined “username”, “Email”, and “password” in a GitHub repository. This is to simulate a database and make API calls during user authentication. The second instance involves storing detailed information about drugs in a GitHub repository, reading it during searches. This approach is preferred over storing the information in a local file since, with an increase in the number of drugs, frequent updates to the version would be required.

I will showcase the relevant steps of API operations by calling the drug information API. First, maintain relevant variables in “this.state” to record the fetched information, fetching status, and potential error messages. The variables "loading" and "error" will provide feedback for users and error correction, which aligns with UI design principles.

```
        this.state = {
           // Other state variables
            drugs: [], filteredDrugs: [], loading: true, error:"",
        };
```

Then, through the “componentDidMount” method, which allows React to indicate to the calling environment that the current component has mounted successfully, the actual operations are executed. The “fetch” function is used to initiate a network request to a specified URL. Once the request is successful, the “response.json()” method is used to parse the response body into JSON format, and the “setState()” method is employed to update the state variables' status. If the network request fails, the “.catch” section captures the error. The error message is logged to the console and stored in the component's state; in both cases, the "loading" is set to false.

```
    componentDidMount() {
        // Other codes
        fetch('https://raw.githubusercontent.com/Jeffrey-Gordon/drug_API/main/drugs.json')
            .then(response => response.json())
            .then(data => this.setState({ drugs: data, filteredDrugs: data, loading: false, error: ""}))
            .catch(error => {
```

```
            console.error("Error fetching data:", error);
            this.setState({ error: "Error fetching data" });
            this.setState({ loading: false });
        });
  }
```

### 3.1.6 Input Validation

Input validation is used in Clock page when user try to add an alarm tip of taking medicine. Before my enhancements to the code on Clock page, it already included checks to ensure that when the user clicks the navigation button, it verifies whether the entered hour is less than 24 hours, and the minute is less than 60 minutes. However, there are still a lot to be improved. The following validations are what I added.

If the user did not login, alert jumps out and viewport is routed to the Login page.

```
  const loggedInUser = localStorage.getItem('loggedInUser');
  if (!loggedInUser) {
    alert("Please log in to add the alarm.");
    history.push('/login');
    return;
  }
```

If the user hasn't provided complete information, prevent the navigation, and display a prompt.

```
  if (!hour || !minute || !selectedOption) {
    alert("Please fill in all required fields");
    return;
  }
```

If the user enters a single-digit time, add a leading zero at the beginning.

```
  const addLeadingZero= (value) => {return value<10 ? "0" + value : value;};
  const newItem = {
    // Other codes
    hour: addLeadingZero(hour), minute: addLeadingZero(minute),
  };
```

### 3.1.7 Filter Operations

Filter operation is used in the Homepage section to search for the desired medication and supports “popular search tags”. The “handleSearch()” method takes user input data “searchTerm” and matches it with the “drugs” data using the “filter()” method. The “DrugName”, “Indications”, and “ActiveIngredient” properties from the previously fetched “drugs” data are transformed to lowercase using the “toLowerCase()” method. They are then individually compared with the “searchTerm”, which is also converted to lowercase. If any of these properties contains the “searchTerm”, the data is returned to a new array for output.

```
handleSearch = (searchTerm) => {
  const { drugs } = this.state;
  const filteredDrugs = drugs.filter(drug =>
    drug.DrugName.toLowerCase().includes(searchTerm.toLowerCase())||
    drug.Indications.toLowerCase().includes(searchTerm.toLowerCase()) ||
    drug.ActiveIngredient.toLowerCase().includes(searchTerm.toLowerCase())
  );
```

```
  this.setState({ searchTerm, filteredDrugs });
};
```

Additionally, the handleSearch() method can also be used to support "popular search tags". I maintain an array in this.state to simulate popular search tags, which are rendered below the search bar. When these tags are clicked, the handleSearch() is invoked to search for the content within the clicked tag.

```
    handleButtonClick = (index) => {
        this.setState({ activeButtonIndex: index });
        this.handleSearch(this.state.popularSearches[index]);
    };
```

### 3.1.8 Sort Operations

I assisted the team member responsible for the Calendar page in sorting the alarm clock tips based on the medication time. In the following code, I compare and sort all elements in the "items" array by comparing the sum of minute value and product of hours multiplied by 60.

```
  items.sort((a, b) => {
    const timeA = parseInt(a.hour) * 60 + parseInt(a.minute);
    const timeB = parseInt(b.hour) * 60 + parseInt(b.minute);
    return timeA - timeB;
  });
```

## 3.2 Implementations in Others Work

I have implemented similar functionalities to most of the features implemented by other team members. Listed functionalities bellow are which did not appear in my work.

### 3.2.1 Presenting Options to Users

The Clock page utilizes a Dropdown component for user options. It takes an array of options, displays a dropdown menu when user open the options, and updates the selectedOption state upon user selection.

```
{isOpen && (<ul className={styles.dropdownlist}>
 {options.map((option, index) => (
   <li key={index} onClick={()=>handleSelectOption(option)}>{option}</li>
 ))}
</ul>)}
```

### 3.2.2 Increase/Decrease Numerical Values

Call the following function when the button is pressed.

```
  const handleNumDecrement = () => {setNum(num - 1);};
  const handleNumIncrement = () => {setNum(num + 1);};
```

# 4 Overall Evaluation

## 4.1 Simple Acceptance Testing

I am satisfied with most of the features of this application. The application has implemented responsive layout, achieved most of the expected functionalities, and adheres to UI design principles. I mainly used the developer tools of the Edge browser and the site *Responsinator* for testing.

## 4.1.1 Responsive Design

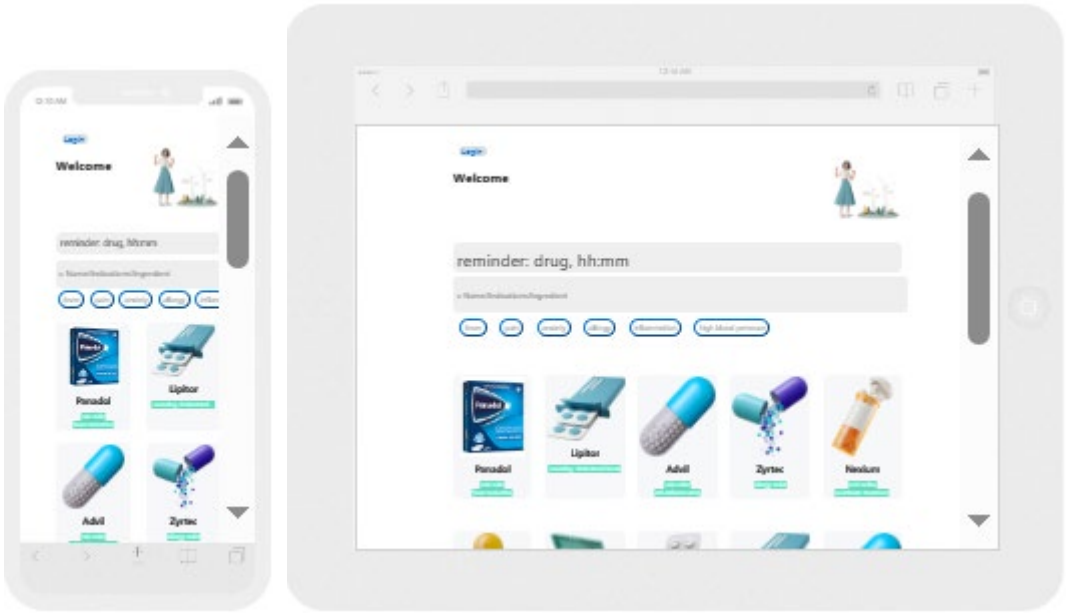


Figure 2 Grid Layout of Home page


Figure 3 Fluid layout of Login page

## 4.1.2 Input Validation and Auto-Completion

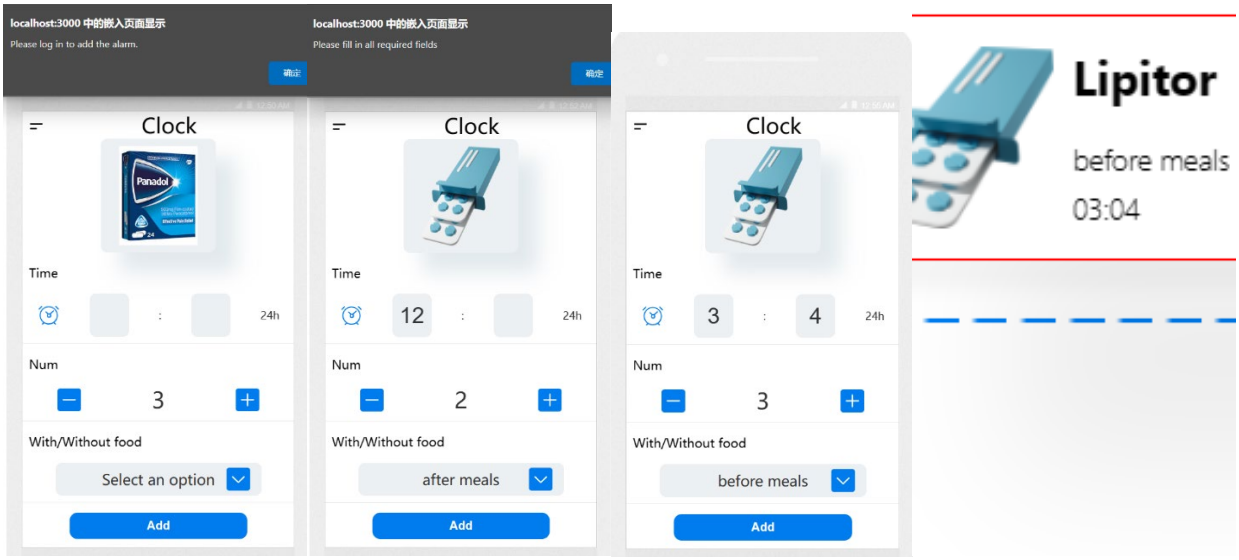


Figure 4 Input Validation and Auto-Completion

## 4.2 Dissatisfaction

As of the project deadline, there is still an unresolved issue - the inability to correctly remove the alarm clock tips from the calendar. Due to the tight timeline and the poorly structured data structure by the member responsible for the Calendar page, the code was difficult for other team members to understand and maintain. Therefore, the member responsible for the Calendar page adopted a temporary solution, using CSS styles to hide the corresponding data when the delete button is clicked. However, when users switch to the Home page and then switch back, the deleted alarm clock tips reappear.

I have the following speculation about the cause of this error: The member responsible for the Calendar page maintains a state variable called "detail." However, the "detail" derives from "allData" variable, and the "allData" derives from "items" variable obtained from localStorage. The member's code only changes the value of the "detail" variable but does not modify the corresponding value stored in localStorage. This results in the restoration of the "detail" variable's value when the page is re-rendered.

```
// items
  const storedItems = localStorage.getItem('calendarItems');
  let items = JSON.parse(storedItems);
  items = items ? items : [];
  localStorage.setItem('calendarItems', JSON.stringify(items));
// allData
  const allData = {4: items.map((item) => ({ item: [{
    id: item.id, drugName: item.drugName, done: false, capsule: item.capsule,
    hour: item.hour, minute: item.minute, image: item.image
  },],})),};
// detail
  const [detail, setDetail] = useState([]);
  useEffect(() => { setDetail(allData[curDate] ?? []);}, [curDate]);
//handle method
  const handleDelete = (id) => {
    let newDetail = detail.filter((timeSlot) => {
      timeSlot.item = timeSlot.item.filter((drug) => drug.id !== id);
      return timeSlot.item.length > 0;
    });
    setDetail(newDetail);
  };
```

## 4.3 Changes to make if time is sufficient

First, the delete functionality should be fixed. Second, an option to select the date should be added to the Clock interface, allowing users to plan medication times for tomorrow and the day after etc. Third, using “localStorage” to maintain account states lacks some security, so let’s consider implementing JSON Web Tokens (JWT). Fourth, adding a backend database for this application is recommended. Fifth, while the Home and Login pages I am responsible for have implemented responsive design, there is room for improvement. Additionally, the Detail page does not currently have responsive design and could benefit from an upgrade.