**Methodology brainstorm**

Stage I:

We need to develop something that contains IBM chatbot and AR capabilities. After consideration, we intended to develop a website because people can use any devices to access our product easily. The main function of this website is to chat with the IBM chatbot. And the users can obtain information and advice on sustainable lifestyle through the chatbot. However, if only this main function is available on our website, it will be too similar to ChatGPT. And if the users want to cultivate lifestyles and living habits, just chatting with the chatbot is far from enough. Therefore, a routine function which allows users to insist on something sustainable is necessary for our website.

In order to obey coordination and consistency in design rules [reference], we wanted to give users the ability to add routines through the chatbot. In addition, we referred to the routine tools on the market (such as…) and wanted to allow users to customize routines according to their preferences. And we decided to subdivide this function into system routine and user routine so that some routines in the routine function would in line with the sustainable theme.

In addition to the routine function, considering the enthusiasm of users to use the website, we want to build other functions that can give users positive feedback. Referring to two popular software, Forest and Ant Forest, we decided to create a badge system and a tree planting system. And we planned to add AR function to the tree planting system to improve the user experience.

Apart from these functions, we want to create an admin page to allow administrators to manage the system and add tasks. We also want to have an activity page to store Bristol's existing sustainable activities for users to browse so that it will help users to find the activities they want. And it would be better to build a function like ‘Too Good To Go’. ‘Too Good to Go’ is a service with a mobile application that connects customers to restaurants and stores that have surplus unsold food [Reference]. Because it will aggregate information about excess resources, we believe the information can provide some chances for users’ sustainable life.

Stage II:

After a period of development, we determined that routine and chatbot were the main functions. And after users use these functions, the planting trees and badge systems will be opened to motivate users to insist on it.

Since the chatbot can give advice and information of activities, it would be redundant if we built an activity page on our website to give more information about sustainable activities. We decided to remove the activity page from our plan.

As for the ‘Too Good to Go’ page, we also wanted to remove it because we found that there is already a similar application on the market in Bristol. In addition, as a person who wants to have a sustainable life, looking for leftover or expired food is just an option [reference of sustainable lifestyle]. As a user of our website, this function seems to be dispensable. Even without this function, users can find their sustainable lifestyle in other ways through the chatbot.

**Layout Design**

As a website that integrates multiple functions, we decided to build it as a single-page application [reference of single-page application]. One reason is to ensure that the layout of each function of the system is consistent, so that frequent refreshes and switching of web pages will not reduce the user experience. The second reason is that the single-page layout can make the entire page more concise, and we can use repeated parts to help users get started with our webpage faster.

We decided to display different functions to users separately to avoid too much content on a single page. To give user control and freedom, we want to add a navigation bar so that users can choose system functions they want. And the navigation bar will show where the user is.

As for the strategic plan for styling issues, we used styled components package (a library to generate component level styles), Bootstrap and CSS code to adjust the visual aspects of the website.

The styled components packages such as ‘element plus’ and ‘tailwind UI’ can provide us with some mature design patterns. These components follow design disciplines such as consistency, feedback, efficiency, controllability and so on. We believe that we can design the layout based on these components, so that the layout of our website is more in line with modern web design concepts. And, because we can directly use tags to adjust the layout, it can save us a lot of time compared to writing CSS code from scratch. However, styled components only provide limited style options. We want to make modifications to these components to make the layout more consistent with the Glife. So we want to use Bootstrap and CSS in vue.js to modify the style.

Bootstrap [Reference] is one of the most popular CSS frameworks. It is convenient to use Bootstrap to adjust the responsive layout because Bootstrap provides many easy-to-understand and straightforward classes. By using Bootstrap, we can reduce the duplication of CSS code to modify the layout and achieve responsive layout in a relatively short time. In addition, we want to provide better experience for both computer users and mobile phone users. Therefore, we want to design two different layouts separately for both mobile phone and computer. We adopt ‘media queries’[reference], described by MDN docs as “to modify your site or app depending on a device’s general type (such as print vs. screen)”, to show different components according to the media size.

**Implementation**

**Front-end**

Project Structure(去看官方文档)【<https://vue-faq.org/en/development/project-structure.html】>

We built the front-end project in Modular form. It is the most suitable project structure recommended by Vue official documentation [Reference]. Because it divides the whole project into several logical modules that are loosely related to each other, we can develop each module relatively independently. And the low coupling structure is good for development [reference of agile and low coupling structure]. Because we lacked front-end development experience, it is more conducive for us to fragment the project and increase the success possibility of the entire project development. Moreover, Modular project is scalable, which is beneficial to our agile development model.

A screenshot of a computer

Description automatically generated

At the first, our project is divided into logical modules such as initial page and main page. According to the official document, “Each module has its own components, composables, assets, utils; probably api, routes, views, layouts.” The module in our project has only components, assets, router and pages. Instead of setting composables and utils folders, we created stores folder because we used Pinia to manage statement. By using Pinia, we could write Typescript file to store the composition functions in stores folder and these Store structures can be used in components and pages, which is like the logic of building composables and utils.

As for the pages folder, the file in it is more like a page template composed of many single components. So we take these components out of the components folder and put them separately in the pages folder. We also applied simple atomic design to components structure. The hierarchical structure based on component complexity makes the structure of components scalable and organized [reference of atomic design]. And we categorized all the components and place them in correspond subfolders of components folder so that retrieval and browsing are easier. In addition, rather than using redundant atoms, molecules and organisms in each subfolder of components folder, we only created some specific folders for reusable components. The rationale behind this decision is that our project is a small-scale initiative with a constrained development timeline. Given that the front-end design is relatively straightforward and does not involve extensive reuse of components, we implemented the structure previously mentioned.

The router folder is just like the routes folder mentioned in official documentation. As a single-page website, we need router to change the content in the page. So we put the routing configuration file and the file containing the routing paths to be protected in this folder. The assets folder is used to store some static resources that need to be preloaded, such as tree images, badge images, and some icons.

Front-end intro:

The front-end is responsible for the layout and styling of website, and controls how users interact with the UI [ G. Blake, Importance of css: what it means why it matters, Apr 2019. https://www.tentononline.com/importance-of-css/ ]. We divided the front-end work into modules according to their functions. [TODO: need to add something]

**1**. Routes

1.1 Routing structure

A diagram of a login form

Description automatically generated

The routes of our website is shown in the figure above[…]. We chose createWebHistory as the history mode because it provides clean URLs and full browser navigation [reference of createWebHistory]. The routes are divided according to the login-register system, user system and admin system. Within each system, sub-routes are further divided by functionality.

After users navigates to the URL of our website, they will be redirected to the welcome page. Users can interact with the UI on the welcome page to enter the login-register system. In the login-register system, we did not set up route switching.

After users log in to the user system, they will be directed to the main page, with the child route redirected to the dashboard. There is a navigation bar to facilitate switching between four components and their corresponding routes in the main page. If users log in to the admin system, they will be directed to the admin page, with child route redirected to the random task and invoke the corresponding components. Admin page also has a navigation bar for switching between two functionalities. In both the user and admin systems, the log out function will direct users back to the login-register system.

1.2 Route Security and Access Control

A screen shot of a computer code

Description automatically generated

To ensure the security of route paths, the website prevents users from directly accessing the main page or admin page via specific URLs before logging in. Before each route change, we first check whether the target URL is a protected URL. And we also verify the user’s login status using information stored in Pinia. Only logged-in users are allowed to access protected URLs, otherwise, they will be redirected to the login page.

**2**. Pages and Components

**1**. Welcome page

UI design

**2**. Login-Registration page

Since we need to display correspond information to different users, we need a login function which used to obtain and verify the user’s identity. And a register function is also required.

2.1 UI Design(或者把uidesign加到每一个单独的组件里去？)

登录：

两个输入框，一个toggle供用户选择登录管理员还是用户系统（因为想增加consistency，不想再多加重复组件），一个button，一些提示信息, 还有tutorial

注册：

强调下与登录的不同点（verification code）

2.2 Implementation

Login-Registration page is composed of the login component and the registration component. Below, we will introduce their implementation one by one.

2.2.1 Login

A diagram of a program

Description automatically generated

The program flow is shown in the figure [???]. Whenever the user clicks the login button, the method bound to the button will be triggered. In this method, the validity of the form is first checked. If the user has not entered a valid username and password, a prompt message will be shown on the website. Because the admin login interface and user login interface are actually the same component, the next route path and the API request path will be determined based on the user’s choice if the form is valid. Afterward, we use Axios to send an API request to the backend with the username and password information. If the response is successful, it indicates that the user has the right to access the system.

Before switching to the next page and displaying the login success message, we use a Pinia file to store the user information returned from the backend. This includes multiple details such as the username, user ID, user permission, and whether the user is new, to facilitate the use of this information in subsequent functionalities. In addition, if the user wants to log in to the user system, we will use another Pinia file to create and store the WebSocket. We use WebSocket to enable two-way communication between frontend and backend. Its usage will be discussed later in the context of the random task in chapter[???]. [https://www.rfc-editor.org/rfc/rfc6455.html]

2.2.2 Registration

The program flow of the registration component is similar to that of the login component. The difference is that the registration component includes a feature for verifying the validity of the email address.

A diagram of a program

Description automatically generated

As the flow shown in the figure [???], the frontend will send an API request to the backend containing the email address after validating the email format. This API request prompts the backend to send a verification code to the target email address and perform the corresponding record-keeping operations. To prevent users from requesting the verification code multiple times within a short period, we disable the button for one minute after confirming the response of the ‘sendCode’ API request is successful. Since we want the button to remain disabled within the specified time even after the user refreshes the browser, we use cookies to store the end time of the countdown.

A computer screen shot of code

Description automatically generatedA screen shot of a computer code

Description automatically generated

We record the expiration time in the cookie by adding the current time to the countdown value. And we use the setInterval function to update the value of countdown every second, allowing it to be displayed to the user dynamically. In addition, we try to fetch the data in the cookies every time the registration component is mounted. Therefore, the countdown value is preserved correctly whether the user refreshes the page or switches components between login and registration.

2.2.3 The switching between login and registration

A black background with white text

Description automatically generated A black screen with text and symbols

Description automatically generated

As we said in the routes section, there is no route switching in the login-register system. Instead, we use v-if and a reactive variable on this page to toggle the display of the login and register components. We have two reasons for this approach. Firstly, the login-register system only involves two components and has no functional expansion. Consequently, we believe it is unnecessary to manage URL switching via routing, as this simplifies state management. Secondly, this method allows for instant component switching by toggling the visibility of components. It avoids the overhead of mounting and unmounting components [Reference of difference between routing and ??], thus preventing re-rendering operations during the switch which may diminish the user experience.

3. Mainpage

3.1 The UI design of the main page

The main page of the user system is composed of a navigation bar and a router view. 这样做的目的。 导航栏以及其实现，其他部分结合路由跳转来切换对应组件的挂载与渲染… 还有对手机和电脑端的分别适配（讲一下为什么要这样做，大概长啥样，）

Tutorial(根据客户的建议，为了使用户能更快的上手并熟悉我们网站的功能，我们做了一个针对新用户的tutorial。)

3.2 Implementation （UI的implementation）

3.2.? Tutorial

We used the Element Plus tutorial component to set up targeted tutorials for each functional page, explaining how to use the features on current page. Since the tutorial is only displayed to new users, we used data from the Pinia file, which stores user information, to determine if the user is new and accordingly enable the tutorial for each page.

Need to add content here…

3.3 Chatbot

3.3.1 UI design and demo

To create a chat interface that enhances the user experience, we drew inspiration from the UI designs of popular chat tools on the market, such as WeChat and WhatsApp. The entire interface is composed of two parts: a chat window and an input box located at the bottom.

A screenshot of a chat

Description automatically generated A screenshot of a chat bot

Description automatically generated

When a user sends a message, user’s message will be displayed on the right, while the chatbot’s replies will be displayed on the left. To further distinguish these messages, we set the user messages with a light green background and the chatbot replies with a grey background.

Additionally, because our chatbot not only replies with text messages but also uses options as responses. When the chatbot replies with options, we display them as interactive buttons within the chatbot's message bubble. The style of the buttons is based on Flowbite’s gradient monochrome. Users can click these buttons to send the corresponding message directly. This enhances the interactivity of the chat system.

Furthermore, like chat applications, we will automatically scroll the chat window down to display the latest messages after the chatbot responds. It ensures that users can easily view the most recent content.

In some chatbot application such as ChatGPT, there is a gradual text generation effect during the chatbot's responses, which creates the impression that the bot is genuinely thinking and engaging in conversation with us. So, we added a loading effect before displaying each message of the chatbot. The loading effect is based on spinner indicator of Flowbite [https://flowbite.com/docs/components/indicators/#loading-indicator]. We believe this will provide a better user experience compared to directly displaying the chatbot's messages.

A screenshot of a chat

Description automatically generated

Moreover, to avoid displaying an overly empty interface when there are no messages, we created an initial window that includes a greeting and two interactive options. It is displayed when users first log in. It serves to guide users, helping them quickly get accustomed to the chatbot feature. Additionally, we hope it indicates that the chatbot's responses are interactive and can be clicked.

3.3.2 implementation

3.3.2.1 useChatStore Pinia file

We wanted to store all the chat messages exchanged between the user and the chatbot during current session for the user to review. Considering that chat messages will be used across multiple components, we created a Pinia file for simpler state management.

A screen shot of a computer code

Description automatically generated 

We need to store the sender and type of each message to facilitate using different UIs and components for displaying various types of messages later on. The ‘messages’ is a reactive array used to store ‘ChatMessage’ objects. Therefore, the messages on the webpage can be dynamically updated when we push a new message in it. And since all messages in the array are stored in chronological order, we can use v-for to iterate through all the messages sequentially.

3.3.2.2 UI implementation

A diagram of a message list

Description automatically generated

The display of InitialWindow is controlled by a variable stored in the Pinia file. The default value of the variable is true. Since we consider user button clicks as sending messages as well, in the method triggered by sending a message, we set this value to false to hide the component. This approach ensures that after the user clicks a button or sends a message, the initial window is closed, and the message list is displayed.

The style of the message bubbles is based on the default chat bubble style from Flowbite. [https://flowbite.com/docs/components/chat-bubble/] We retained only the avatar, username and message bubble frame from their design. And we added dynamic classes to the corresponding component tags to achieve the styles mentioned before.



For example, in the code snippet shown above, we use the information stored in the Pinia file. If the message sender is the user, we will add the flex-row-reverse class to it. Therefore, the content of the user message bubble will be laid out in reverse order (right to left in an LTR layout).

A screen shot of a computer program

Description automatically generated

In addition, we use flex to display message bubbles and use align-self to align them either to the left or right. By adding a maximum width, we achieve the effect where user messages are aligned to the right and bot messages are aligned to the left.

A computer screen with text

Description automatically generated

A computer screen with text

Description automatically generated

Regarding the loading effect, in addition to checking if the message is a reply from the bot, we also need to use a variable to determine whether the loading effect should be rendered. The default value of the ‘isLoading’ is false. We will set its value to true for one second when each message bubble component with a sender of bot is mounted, to implement a one-second loading effect.

A screenshot of a computer

Description automatically generated

The message type is used to determine whether the UI should display the message as a text message or as a clickable button.

还有啥？(需要在这里补充)

3.3.2.3 Program flow

A diagram of a message

Description automatically generated

After the user sends a message, we add this message to the messages array in the Pinia file so that the latest user message will be displayed in the UI. To get the response of the IBM chatbot, we send a post API request to the backend with the message string. After that, if the response is successful, the handleResponseData() method will be called.

A screen shot of a computer code

Description automatically generated A black screen with colorful text

Description automatically generated

This method parses the response data and adds the messages to the message list. Since the backend replies with a variable number of messages contained within an array, we use forEach() to add them one by one. This ensures that the order of the messages is preserved correctly when displayed on the frontend. Moreover, since messages are displayed immediately on the frontend UI after being added, we set a timeout within each iteration of the forEach() loop to manually control the time interval for adding messages. This creates a front-end effect where a new message appears every second.

3.3.2.4 The connection with the routine functionality

Before sending the API request to the backend to get chatbot’s response, there is a step to check whether the user's input is a period word. If the user's input contains one of "daily", "weekly" or "monthly", the addSystemRoutine() function will be called to add the selected routine to the user’s routine list. Although users might include these three words in other messages, the backend has appropriate handling logic, which is why we implement such a simple logic in frontend. In addSystemRoutine() function, we convert the detected period word into the format required by the backend and send an API request to it.

3.4 Routine

3.4.1 UI design and demo

A screenshot of a computer

Description automatically generated A screenshot of a phone

Description automatically generated

To categorize and display the user's routines by period, we used tabs and placed the categorized routines in their respective tabs. The tab styles are inspired by Flowbite's tab component. [https://flowbite.com/docs/components/tabs/] Using tabs to categorize routines is similar to a navigation bar. It allows users to clearly understand the current content displayed by the system and enables them to freely switch to the tab they wish to view.

Moreover, we assigned different background colours to system routines and random tasks to distinguish between user-defined routines, system routines, and random tasks. This allows us to place all three types of routines in a single list, making the UI more streamlined. The different background colours also remind users that completing routines with coloured backgrounds earns them points for tree planting. When users complete a routine and check it off, the routine will have a strikethrough effect, making it more intuitive to distinguish from incomplete routines.

Screens screenshot of a phone

Description automatically generated

To maintain consistency in the page, we implemented the routine-adding feature as a toggle modal component which is inspired by Flowbite’s Modal component. [https://flowbite.com/docs/components/modal/] It visually separates the routine list and the adding functionality, making it look more aesthetically pleasing.

3.4.2 Implementation

3.4.2.1 useRoutine Pinia file

 A screen shot of a computer code

Description automatically generated

Similar to the implementation of the chatbot functionality, we use Pinia files to store data and functions for all components related to routines. We created two separate files for system routines and user customized routines. And we parse all routines obtained from the backend and store them as Todo object. When we need to use the data, we can retrieve it directly from Pinia, avoiding the need to send an API request each time. Besides, we also store data such as the currently active tab and the tabs list to manage the display of corresponding routines.

3.4.2.2 UI implementation

A screenshot of a computer program

Description automatically generated

To implement the Routine List UI mentioned in design part, we performed two rounds of filtering on the todos stored in the Pinia file. The first filtering step stores different types of routines separately. We fetch user routines and system routines from two different API endpoints. We then use the ‘type’ attribute of routines to differentiate between random tasks and system routines because they are stored together in the backend. Finally, we filter the routines based on the currently active tab to achieve the UI design of the routine list. Besides, we use the id of each Todo to adjust the order in which routines are displayed in the UI, ensuring that the most recently added routine appears at the top of the routine list.

3.4.2.3 Interaction with the backend

Each time the routine component is mounted, we send an API request to the /api/routine/init to retrieve all routines for the user. To ensure that routines are updated in the database in real-time after being edited, added, or deleted on the website, we send API requests to the corresponding endpoints. After using these features, we also make another request to the /api/routine/init endpoint to fetch the latest data.

3.5 Dashboard

3.5.1 UI design and demo

Based on feedback from clients and supervisors, we need to add some components to the dashboard to visualize data, making it easier for users to browse. Therefore, we placed the routine completion status and the current number of trees planted as two separate components on the dashboard. This allows users to easily view their current progress directly from the dashboard.

Besides, we included an interactive component for adding random tasks in the dashboard. The reason for this is that having interactive components in a comprehensive dashboard does not disrupt the overall consistency. Moreover, the content of this component is not substantial enough to be presented as a standalone page.

A screenshot of a computer

Description automatically generated

3.5.2 Implementation

3.5.2.1 UI implementation

These three components are displayed using a grid layout, which allows us to easily define their respective sizes and areas. For routine status, we wanted to use progress bars to display the user's routine completion status, as it provides the most straightforward visualization of such data. We used Element Plus's Progress component to display the user's routine completion status. [https://element-plus.org/en-US/component/progress.html] Due to time constraints, we used only Vue's reactive variables to display text in the other two components.

3.5.2.2 Random Task

A black screen with white text

Description automatically generated

In the previous login system, as we mentioned, the frontend will connect to the backend via WebSocket before the user log into the user system and switch routes. To display the string sent by the backend via WebSocket and the remaining task count, we used two reactive variables to store these pieces of data in the frontend. When the WebSocket receives information, it updates the task content and automatically invokes a method to send an API request to the backend to retrieve the remaining task count. Additionally, we employed a computed property to handle the display of a message when the remaining task count reaches zero.

4. Admin page

4.1 UI design and demo

A screenshot of a computer

Description automatically generated

We used a similar layout on both the Random Task and Admin List pages. This layout was inspired by the list component layout from Flowbite [<https://flowbite.com/docs/typography/lists/>], which allowed us to implement the frontend UI quickly. Another reason for this approach is that the functionality of these two pages is similar. Therefore, we only needed to modify the form's content and the displayed information.

4.2 Implementation

To display the admin list and the history of random task posts, the frontend sends API requests to the backend each time the corresponding page is mounted to retrieve the data. After parsing, the data is stored in the corresponding Pinia file on the frontend. For the publishing feature and adding a new admin, we modelled the method and layout used for adding routines in the routine section, creating a toggle modal. When the user clicks, the corresponding form is displayed to the user. After the user submits the form, the frontend sends an API request to the backend to process the data, followed by another request to update the list with the latest data.

个人报告写出

关于项目的想法，想这样做想那样做，但是为什么做了为什么没做，好的坏的都要写