

**UECS3253/UECS3463 WIRELESS APPLICATION DEVELOPMENT**

**Book Rental Application**

|  |  |  |  |
| --- | --- | --- | --- |
| **Group Name: G33** | | | |
| **Student Name** | **Student ID** | **Practical Group** | **Programme (SE/ET)** |
| **E Zhong Lin** | **2001777** | **P4** | **SE** |
| **Lee Boon Hao** | **2106860** | **P1** | **SE** |
| **Ch’ng Jay Sheng** | **2004911** | **P1** | **SE** |
| **Kim Yuan Yuan** | **2005317** | **P1** | **SE** |

**Lee Kong Chian Faculty of Engineering and Science**

**Universiti Tunku Abdul Rahman**

**TABLE OF CONTENT**

[INTRODUCTION 1](#_Toc145367001)

[KEY FEATURES 1](#_Toc145367002)

[Part A 3](#_Toc145367003)

[A SCREENS, COMPONENTS AND REACT-NAVIGATION 3](#_Toc145367004)

[A.1 Screens & Layout 3](#_Toc145367005)

[A.2 Use of Components 22](#_Toc145367006)

[A.3 Reusability (Custom Components) 28](#_Toc145367007)

[A.4 Implementation of React-Navigation 38](#_Toc145367008)

[Part B 41](#_Toc145367009)

[B DATA PERSISTENCE 41](#_Toc145367010)

[B.1 Usage of Data Persistence 41](#_Toc145367011)

[B.2 Implementation of Data Persistence 47](#_Toc145367012)

[B.3 CRUD Operations & Input Validation 56](#_Toc145367013)

[Part C 70](#_Toc145367014)

[C CLOUD CONNECTIVITY 70](#_Toc145367015)

[C.1 Usage of Cloud Connectivity 70](#_Toc145367016)

[C.2 Implementation of Cloud Connectivity 74](#_Toc145367017)

# INTRODUCTION

A book rental app is a mobile application or web platform that allows users to borrow or rent books for a specific period of time, typically for a fee or subscription. These apps provide a convenient and cost-effective way for readers to access a wide range of books without having to purchase them outright.

In this assignment, we are trying the develop a book rental application which consists of functions such as book list management, wishlist management, cart and order management, contact us features and the admin book management system. By using our application, users or clients can access our application to view the booklist which is stored in our database and try to proceed the action to rent the book easily. Users can directly choose the book and look around all the descriptions of the book and press some button to rent our books successfully. All of the features are user friendly and easy to manage, most of the users will not face problems when using this app.

Other than that, we are having a database function to store all the book information and user information in our database. By using a database, all of the action and information can be managed well to avoid any confusion. Mostly, the database we will use to store all the book relevant information such as name, author, description and admin can use this information to manage and apply to show to users. By having a database, all the action can be maintained smoothly.

Lastly, we are trying using the Web-socket Api and Web-based Api to let our client have a request to use all of the features of book application online. By using these two API, all the client can manage their action and admin also can maintain the application action all in well and smoothly.

# KEY FEATURES

|  |  |
| --- | --- |
| **Features** | **Description** |
| Book list management | The user can view all the books available from us. The user can also search for a book to find a book that they like. They can also add the book to the wishlist and cart accordingly. They can also view the books with its details. |
| Wishlist management | The user can view all the books that are being added to the booklist. They can also remove the books from the booklist. |
| Cart and order management | The user can view all the books that are being added to the cart. They can also remove the books from the cart. They can also place an order by filling up their order information. After placing an order. they can view back their order. |
| Contact Us features | The user can fill in the contact us form in order to contact us. |
| Book management | The admin can login to the app and create, view, delete, or update the book details. |

# 

## SCREENS, COMPONENTS AND REACT-NAVIGATION

### Screens & Layout

1. Screen 1: Home Screen

A screenshot of a phone

Description automatically generated

The home screen in here will allow the user to navigate to the other six screens: UserBookList screen, Wishlist Screen, Cart screen, OrderHistory screen, Search screen, ContactUs screen, and AdminLogin screen.

1. Screen 2: UserBookList Screen

A screenshot of a book store

Description automatically generated

This screen will show all the books which are stored inside the database. The back button at the top left of the screen will direct the user back to Screen 1: Home Screen. It links with the function of search book, add to cart and add to wishlist function. If the user presses the books, it will redirect to Screen 3: UserView Screen.

1. Screen 3: UserView Screen

A screenshot of a phone

Description automatically generated

In this screen, it will show more details of books including name, category and author of the books which is recorded manually by admin in the database. There is a back button on the top left of the screen which the users can press to go back to Screen 2: UserBookList screen. The title of the screen will also change to the book’s name that the user taps. It also has the “Add-to-cart” and “Add-to-wishlist” button that allows the user to add the book to cart or wishlist. These two buttons have the same functionality with the two buttons in Screen 2: UserBookList Screen.

1. Screen 4: Wishlist Screen

A screenshot of a book

Description automatically generated

In this screen, it will show all the books that are added to the wishlist. There is a back button at the top left of the screen that allows the user to go back to Screen 1: Home screen. The user can press the books to redirect to Screen 3: UserView screen. Besides that, the user can press the trash button at the right side of each book in order to remove it from the wishlist.

A screen shot of a bookmark

Description automatically generated

If the user has not added any books to the wishlist, the screen will show the messages of “No books added to the wishlist yet”.

1. Screen 5: Cart Screen

A screenshot of a book

Description automatically generated

This screen will show all the books that are added to the cart by the user. There is a back button on the top left of the screen that allows the user to go back to Screen 1: Home screen. The user can press the book to direct to Screen 3: UserView screen. There is a text at the bottom left of the screen that will show the amount of the books to be rented. There is also a button named “Place Order” on the bottom right of the screen that will pop out a confirmation message to the user before directing the user to Screen 6: Order screen. Besides that, the user can press the trash button at the right side of each book in order to remove it from the cart.

A screenshot of a phone

Description automatically generated

If the user has not added any books to the cart, the screen will show the messages of “No books added to the cart yet”.

1. Screen 6: Order Screen

A screenshot of a phone

Description automatically generated

This screen will allow the user to enter his/her information before renting the books. There is a back button on the top left of the screen which will direct the user to Screen 5: Cart screen. The user needs to enter his/her name, email and phone number. The screen will also show all the books that the user wants to rent. After the user enters the correct information and presses the “Finish” button, orders will be placed and the user will be redirected to screen 1: Home screen.

1. Screen 7: OrderHistory Screen

A screenshot of a phone

Description automatically generated

This screen will show the rental that the user has done. The back button on the top left of the screen will redirect the user back to the Screen 1: Home screen. The screen will show the order information that the user fills in the Screen 6: Order screen. It will also display the book that has been rented by the user. There is a “Clear Order History” button below of the books rented that allows the user to clear the order history.

A screen shot of a phone

Description automatically generated

If the user has not made any order, there will be a message displayed on the screen which is “No rental history”.

1. Screen 8: Search Screen

A screenshot of a video game search

Description automatically generated

In this screen, the user can enter the keyword to search the books. There will be a back button on the top left of the screen that allows the user to go back to Screen 1: Home screen. For example, the user inputs a ‘hi’ keyword and the database will show all the books which consist of the ‘hi’ keyword. The user can also perform the “add-to-cart” function and “add-to-wishlist” function by pressing the “Add to cart” and “Add to button” buttons. The user can also press the books and direct to Screen 3: UserView screen.

A screenshot of a search engine

Description automatically generated

If the user enters the keyword that does not have any results or if the user presses the “Clear” button, there will be a message shows “No results found”.

1. Screen 9: ContactUs Screen

A screenshot of a contact us form

Description automatically generated

In this screen, the user can contact us by filling up this form. After filling up this form, the user will see a modal that will tell them whether they have successfully submitted the form.

1. Screen 10: AdminLogin Screen

A screenshot of a login screen

Description automatically generated

In this screen, the admin can login by entering the correct name (“admin”) and password (“123454321”). There is a back button on the top left of the screen that will direct the user to Screen 1: Home screen. If the user enters correct name and password, the user will be direct to Screen 11: AdminDashboard screen.

1. Screen 11: AdminDashboard Screen

A screenshot of a book list

Description automatically generated

This screen will show the extra functionality of admin. There are two buttons on this screen. The first button “Book List” will direct the admin to Screen 12: BookList screen and the second button “Logout” will direct the admin to Screen 1: Home screen.

1. Screen 12: BookList Screen

A screenshot of a phone

Description automatically generated

This screen will show all the books in our database. There is a back button on the top left of the screen that will direct the admin to Screen 11: AdminDashboard screen. Each book will have an image, category, name, and author. There will be an add icon image, on the bottom right of the screen that will direct the admin to Screen 13: Create screen. The admin can also press the book to direct to Screen 14: View screen.

1. Screen 13: Create Screen

A screenshot of a computer

Description automatically generated

This screen allows the admin to add a new book to the book list. There is a back button on the top left of the screen which will direct the admin to screen 12: BookList screen. The admin needs to enter the book’s name, image, and author. Besides that, the admin will also need to choose the category of the book.

After the admin enters the correct information of the book and presses the “Save” button, a message of “Added successfully” will pop out and the admin will be direct to Screen 12: BookList screen and the admin will see that the book has been added to the book list at Screen 12: BookList screen.

1. Screen 14: View Screen

A screenshot of a phone

Description automatically generated

This screen will show the details of the particular books. There is a back button on the top left of the screen and it will direct the admin to the Screen 13: BookList screen. This screen will show the book’s image, name, category, and author. There is an add button at the bottom right of the screen.

A screenshot of a phone

Description automatically generated

After the admin clicks on the add button, the admin can choose whether he/she wants to delete / edit the books. If the admin chooses to edit the book, he/she will be redirected to the Screen 15: Edit Screen.

A white background with black text

Description automatically generated

If the admin chooses to delete the book, a pop-up message of “Confirm to delete” and the book’s name will be displayed and the admin can choose whether he wants to confirm the deletion of the book. If he chooses “no”, nothing will happen and the admin will be directed to this screen again.

If he chooses “yes”, the book will be deleted. A message of “Deleted successfully” will be popped up and the admin will be directed to the Screen 12: Book List screen.

1. Screen 15: Edit Screen

A screenshot of a phone

Description automatically generated

This screen will allow the admin to update the book details. The admin can update the book’s name, category, image, and author. There is a back button on the top left of the screen and will direct the admin to the Screen 14: View screen if the admin presses it. There is also a button named “Save” on the bottom of the screen.

A screenshot of a phone

Description automatically generated

After the admin updates the book details, for example: he changes the author from “Woolf Virginia” to “Woolf Virgini”. A message of “Record UPDATED for” and the book’s name will pop up and the book details will be updated.

### Use of Components

1. Third party-components::
   1. FlatList from ‘react-native-gesture-hnadler’:

A screen shot of a computer program

Description automatically generated

Figure A2.2a.1: Code snippet for FlatList component used in UserBookList screen

In this code snippet, there are some important props: The first one is data that specifies the array of data that the FlatList should render. Here, it's set to ‘this.state.books’, which is an array containing information about books. The second one is renderItem which is a function that takes an item from the data array and returns a React element to render it. The keyExtractor prop is a function that takes an item from the data array and returns a unique key for that item. The refreshing and onRefresh props are used to implement a “Pull to Refresh” feature. Other props like showsVerticalScrollIndicator and extraData are used to control the appearance and behavior of the FlatList.

* 1. ScrollView from ‘react-native-gesture-handler’:

Component that wraps platform ScrollView while providing integration with touch locking "responder" system.

A screen shot of a computer program

Description automatically generated

Figure A2.2b.1: Code snippet for ScrollView component in UserView screen

In this code snippet, the ScrollView component wraps around the content you want to be scrollable. Inside the ScrollView, you have various UI elements such as Image and InputWithLabel components.If the content within the ScrollView exceeds the available screen space, users will be able to scroll vertically to view the entire content.

* 1. FloatingAction from ‘react-native-floating-action’:

A screenshot of a phone

Description automatically generated

Figure A2.3a.1: FloatingAction component used in View screen



Figure A2.3a.2: Import statement

The figure above shows the floating action component used in the View screen. Firstly, in the View screen, we need to import the statement in order to use the FloatingAction component.

A screen shot of a computer screen

Description automatically generated

Figure A2.3a.3: Code snippet for buttons in FloatingAction

Next, we need to define the buttons that we need to use. In this code, we have defined two buttons, edit and delete button.

A screen shot of a computer code

Description automatically generated

Figure A2.3a.4: Code snippet for implementation of FloatingAction component

Next, we render the FloatingAction component that has three props. The first one is actions={actions} in which this prop passes the array of defined actions to the FloatingAction component. The next one is color={'#a80000'} in which this prop sets the background color of the main floating action button. The last one is onPressItem={name => {...}} in which this prop specifies a callback function that is called when an action is pressed. The name parameter passed to the callback corresponds to the name property of the selected action.

* 1. Ionicons from ‘react-native-vector-icons/Ionicons’:

A screenshot of a bookmark

Description automatically generated

Figure A2.4a.1: Implementation of Ionicons element in Wishlist screen



Firstly, we need to have the import statement in order to use the Ionicons element.



Next, we need to render the Ionicons element. It has three props. The first one is name={"trash"} that specifies which icon to display. In this case, it uses the "trash" icon from the Ionicons library. The second one is size={80} that determines the size of the icon. In this case, it sets the size of the "trash" icon to 80 units. The last one is onPress={() => this.removeWishlist(item)} that specifies an onPress event handler, which is triggered when the user taps on the icon. When the "trash" icon is pressed, it calls the this.removeWishlist(item) method.

1. Styles used
   1. Internal style

A screen shot of a computer code

Description automatically generated

In this code snippet, we have implemented an internal style for the view component in the AdminDashboardScreen. It will have the style of flex 1, and it will also align the child components inside the View component to be centered horizontally by setting the justifyContent to ‘center’. It will also vertically align the child components to center by setting the alignItems to ‘center’.

* 1. StyleSheet: an abstraction similar to CSS StyleSheets

A screen shot of a computer program

Description automatically generated

The code snippet above is the style that is implemented by using the StyleSheet component. We have created a ‘styles.js’ in the same directory as ‘App.js’ and we put all the styles that are implemented by StyleSheet into this .js file.

A screen shot of a computer program

Description automatically generated

If other screens want to implement the StyleSheet, they will need to import the styles.js first before they can implement the StyleSheet.

### Reusability (Custom Components)

1. Custom Components: All these three custom components are created in a .JS file called UI.js
   1. InputWithLabel: This component is designed to create an input field with a label. It supports customization of text input styles, multiline input, and the number of lines. It can be used to create both horizontal and vertical layouts.

A screen shot of a computer code

Description automatically generated

Figure A3.1a.1: Code snippet for InputWithLabel component

The figure above shows the InputWithLabel component inside the UI.js file. The component will have two methods. First is the constructor method. The constructor method is called when an instance of InputWithLabel is created. It initializes the component's properties and state. Here, it checks if the orientation prop is provided and sets the orientation variable to either 'row' or 'column' based on its value. If the orientation prop is not provided, it defaults to 'column'.

Next is the render method. It will render the content inside this InputWithLabel component. It consists of View, Text, and TextInput components. The View component has its styling and the ‘flexDirection’ property of the View component is set to the ‘orientation’ determined in the constructor. The Text component will have the style property that is determined by the ‘textLabelStyle’ prop provided when using this component. The TextInput component represents the input field. It inherits various props like placeholder, value, and event handlers from the parent component using the spread operator {...this.props}. This allows for flexibility in configuring the input field.

A computer screen with text

Description automatically generated

A screenshot of a book

Description automatically generated

The two figures above show the implementation of the InputWithLabel component in the Create screen. The textLabelStyle and textInputStyle specifies the style of the inputWithLabel component, while the label will specify the label text, and the placeholder provides a placeholder text for the input field. value and onChangeText are used to bind the input value to the component's state.

* 1. AppButton: This component represents a customizable button. The button's theme (warning, danger, or primary) by changing its background color. It also supports adding an icon (using Ionicons) and defining press and long-press event handlers.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

The AppButton component will have a constructor. In the constructor, the component sets up its initial state and configuration based on the provided props. It checks the theme prop to determine the button's background color.

Next, in the render method of the component, it will return the actual button view. It uses TouchableNativeFeedback for Android devices to provide touch feedback when pressed. The onPress and onLongPress event handlers are passed through the props, It will also have the styling for the button. If an iconName is provided, it renders an icon using the Ionicons component from the react-native-vector-icons library.

A computer code on a black background

Description automatically generated

A screenshot of a computer

Description automatically generated

The two figures above show the implementation of the AppButton component in the UserBookList screen. The AppButton component will have four props. The first prop is the style that will be applied to the button. The second prop is the title which is the text that will appear in the button. The third prop is the theme which is the background color of the button, and the last prop is the function that the user will be directed to whenever they press the button.

* 1. PickerWithLabel: This component combines a label and a picker (dropdown) input. It allows you to customize the text label style, orientation (horizontal or vertical), and the appearance of picker items.

A screen shot of a computer program

Description automatically generated

The PickerWithLabel component will have a constructor method that will be called when an instance of InputWithLabel is created. It initializes the component's properties and state. Here, it checks if the orientation prop is provided and sets the orientation variable to either 'row' or 'column' based on its value. If the orientation prop is not provided, it defaults to 'column'.

It will also have the render method that will render the content of the PickerWithLabel component. The component renders a View containing a Text element for the label and a Picker element. The label's style and text are determined by the textLabelStyle and label props. The picker items are generated based on the items prop, with each item represented by a Picker.Item.

A screen shot of a computer code

Description automatically generated

A screenshot of a phone

Description automatically generated

The two figures above show the implementation of PickerWithLabel component for Create screen. The props of textLabelStyle and pickerItemStyle will specify the style for the picker. The props for the label is the text that will display as the label for the picker input. The common.states props for the items is an array of items that populates the picker. These items are likely predefined categories for books and these items are stored in the CommonData.js. The mode prop is set to 'dialog', which suggests that the picker should display as a dialog when activated. For the selectedValue prop, it is used to set the currently selected value in the picker. It is controlled by the component's state (this.state.category), so changes in the picker's selection will update the state. The onValueChange prop specifies a callback function to be executed when the user selects an item from the picker. It updates the category state with the selected value.

1. Props and state used in View screen:
   1. Props:
      1. ‘this.props.route.params.id’: This prop is received from the React Navigation route. It represents the id of the book that is currently being viewed.

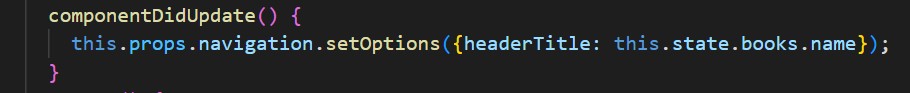
A computer screen with text and symbols

Description automatically generated

* + 1. this.props.navigation: This prop provides access to the navigation functionalities of the screen.



* + 1. this.props.route.params.refresh: This prop is also received from the React Navigation route. It likely represents a function that can be used to refresh data on the parent screen.



* 1. State:
     1. this.state.bookID: This state variable stores the id of the book that is currently being viewed.
     2. this.state.books: This state variable stores detailed information about the book, including its name, category, author, and other attributes.

A computer screen with blue text

Description automatically generated

1. Lifecycle methods (other than render())
   1. constructor(props): The constructor method is a special method in a React component. It will be called automatically when an instance of the component is created. It takes props as its parameter, which represents the properties passed to the component.

A computer screen with text and images

Description automatically generated

This is the constructor method used by the ViewScreen. In this constructor method, it has super(props), which is used to call the constructor of the parent class. For this case, it is Component. It ensures that the parent class's constructor is properly initialized before setting up the component's own constructor. It also has this.state where it will set the data that can change over time. Here, the bookID has been set with the id parameter passed through the component's route. It represents the ID of the book that is being viewed. While the state of the books is initialized to ‘null’, as it will later hold detailed information about the book retrieved from an API call. Lastly, the constructor has method binding. The \_queryByID function is bound to the current instance of the component using .bind(this). This is done to ensure that the function has the correct context when it's called later.

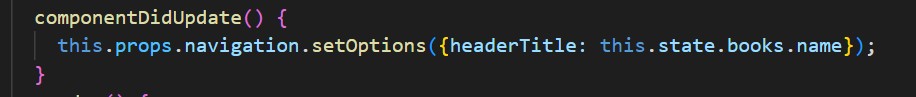
* 1. ComponentDidMount: one of the lifecycle methods provided by React. It is automatically called after the component has been inserted into the DOM.

A black background with white text

Description automatically generated

The figure above shows the ComponentDidMount function used by the View screen. In this function, there is a single method call, which is \_queryByID(), which is responsible for fetching detailed information about the book with the specified bookID.

* 1. ComponentDidUpdate: one of the lifecycle methods provided by React. It is automatically called after a component's state or props have been updated and the component has re-rendered.



The figure above shows the ComponentDidUpdate function used by the View screen. This method call is used to dynamically set the title of the header of the screen based on the ‘name’ property of the ‘books’ state.

### Implementation of React-Navigation

We have implemented stack navigation: Stack Navigator provides a way for your app to transition between screens where each new screen is placed on top of a stack.

A screen shot of a computer screen

Description automatically generated

Firstly, in the first line of this code snippet, we have created an instance of a stack navigator by calling createStackNavigator(), and we store it in the Stack constant. This Stack constant will be used to define and manage the navigation stack in your application.

Next, within the <NavigationContainer>, we define the individual screens and their configuration options:

1. “NavigationContainer” is a component provided by @react-navigation/native that wraps the entire navigation structure. It's the top-level component for navigation in the app.
2. Stack.Navigator is the component responsible for managing the stack of screens. Inside it, we define individual screens using Stack.Screen.
3. Each Stack.Screen represents a screen in the app. We specify the name (a string identifier), the component (the actual React component to be displayed when this screen is navigated to), and options (configuration options for this screen, such as header styling).

A screenshot of a phone

Description automatically generated

A computer screen with text

Description automatically generated

This figure above shows the implementation of the navigation in the Home screen. We have used this ‘this.props.navigation.navigate(screen)’ in order to navigate to another screen. In this code snippet, it will navigate to the ‘AdminLogin Screen’ when we press on the ‘Admin Login’ button.

A computer screen shot of a diagram

Description automatically generated

The figure above shows the level of hierarchy between each screen and the navigation between each screen. Each screen can navigate to the screen on their right level, and go back to the screen on their left level. In the AdminDashboard screen, there is a button called ‘Logout’ where it will navigate to the “HomeScreen” after the user presses the button.

# 

## DATA PERSISTENCE

### Usage of Data Persistence

Data persistence in React Native application development refers to the crucial capability of storing and managing data in a way that it can be retained and retrieved even after the application is closed or the device is powered off. This concept is fundamental for creating robust and user-friendly mobile applications. In React Native, data persistence can take several forms.

Firstly, local Data Storage such as AsyncStorage and React Native SQLite, afford the means to stash away user preferences, configuration settings, and even structured datasets on a user's device. This local reservoir of data not only grants users the luxury of offline access but also ensures that their interactions with the app leave lasting impressions. AsyncStorage is a simple key-value store that can be used to store small amounts of data, such as user preferences or settings. SQLite, on the other hand, offers a full relational database system for more complex data storage needs. Local data storage allows applications to store data on the user's device, enabling offline access and data retention across application sessions. Besides that, caching involves temporarily storing frequently used data, such as API responses or images, locally on the device. This cached data can be quickly accessed, reducing the need to fetch it from a remote server repeatedly. Caching enhances application performance and responsiveness, making it a vital aspect of data persistence in React Native apps. Moreover, React Native relies heavily on component state to manage and display data in the user interface. Component state enables the persistence of data within the scope of a single screen or component, allowing developers to create dynamic and interactive user interfaces. State management ensures that data remains available and up-to-date while the user interacts with the app. For more complex data persistence requirements, such as managing large datasets or structured data, React Native applications can integrate with external databases. SQLite is a popular choice for this purpose, providing a reliable and efficient way to create, read, update, and delete records. Developers can design database schemas to suit their application's needs, facilitating the storage and retrieval of structured data. Furthermore, data persistence is especially critical in scenarios where users need to access application data offline. By storing relevant data locally, React Native apps can continue to function even without an internet connection. This is particularly important for applications like e-commerce platforms, news readers, or productivity tools where uninterrupted access to data is expected. Last but not least, data persistence also plays a role in synchronization. When the application regains an internet connection, it can synchronize locally stored data with a remote server, ensuring that the user's data remains consistent and up-to-date across devices and platforms.

In summary, data persistence is a cornerstone of React Native application development, encompassing techniques and strategies to store, retrieve, and manage data effectively. It empowers developers to create applications that are reliable, responsive, and capable of providing a seamless user experience, even in challenging network conditions. Whether through local storage, caching, state management, or database integration, data persistence is essential for building versatile and user-friendly React Native apps.

**Usage of SQLite**

In our book rental application, SQLite serves as a powerful data persistence mechanism to manage and store various aspects of our application's functionality. It functions as a local relational database that allows us to store and retrieve structured data efficiently.

SQLite in our application is primarily used to:

1. Store Book Information: SQLite is used to store detailed information about the books available for rental. Each book's details, such as name, category, image, author, and its availability status (wishlist, cart, orders), are stored in a structured manner within tables. This enables efficient querying and retrieval of book data for display in our app.
2. Retrieve and Display Data: When users access the application, SQLite is used to fetch data from the database to populate various screens. For example, when a user views their cart or wishlist, SQLite retrieves the relevant book details to display. This enables a responsive and dynamic user interface.
3. User Actions and Updates: SQLite supports user actions such as adding books to the cart, wishlist, or placing orders. When a user performs such actions, the database is updated accordingly. For instance, if a user adds a book to their cart, the SQLite database is updated to reflect this change, ensuring that the user's choices persist across app sessions.
4. Search and Filter Functionality: SQLite plays a crucial role in implementing search and filter functionality. Users can search for books based on keywords, and SQLite executes SQL queries to find matching books efficiently. Filters, such as displaying books by category, are also achieved through tailored SQLite queries.
5. Order History: SQLite helps in maintaining order history. When a user places an order, SQLite is used to update the order status for the selected books. This history can later be queried to show users their past orders.
6. Error Handling: SQLite provides error handling mechanisms to ensure data integrity. It helps manage database connections and transactions, ensuring that the database remains consistent, even in the event of app crashes or errors.
7. Backend Communication: Our Flask backend interacts with the SQLite database to fetch and update data based on user requests. This integration ensures that the data is synchronized between the app's frontend and backend.

In summary, SQLite is a critical component of our book rental application, facilitating data storage, retrieval, and management. It enables the app to provide users with a seamless and responsive experience by persisting essential information and efficiently handling user interactions.

**Usage of AsyncStorage on OrderScreen**In our book rental application, AsyncStorage is used as a local data persistence mechanism to store and retrieve user-related information, such as the user's name, email, and phone number. This allows the application to provide a more seamless and personalized user experience.

When a user enters their personal information, such as their name, email, and phone number, this data is stored securely on the user's device using AsyncStorage. Storing this information locally eliminates the need for the user to re-enter it each time they interact with the app, enhancing convenience and usability.

Later on, when the user proceeds to place an order for books, their stored information is retrieved from AsyncStorage and sent to the server along with the order details. This ensures that the server receives the necessary user data to process the order accurately. After the order is successfully placed, the user's personal information is cleared from both the component's state and AsyncStorage to maintain user privacy.

Overall, AsyncStorage serves as a valuable tool for enhancing user interactions by providing a smooth and personalized experience in our book rental application while securely managing user data locally on the user's device.

**Usage of AsyncStorage on User Order History**In our book rental application, AsyncStorage is utilized as a data persistence mechanism for managing and storing essential user information and settings. It is particularly helpful for preserving user-related data between app sessions. Here's a overview of how AsyncStorage is used in our book rental application:

1. User Data Storage: AsyncStorage is employed to store user-specific information such as their name, email, and phone number. This information is typically stored as key-value pairs, where each piece of data has a unique identifier (key) associated with it. For example, the user's name is stored with the key 'name,' and so on.
2. Data Retrieval: During the application's initialization or relevant screens, AsyncStorage is used to fetch the stored user data. This retrieval process ensures that when a user returns to the app, their previous settings and information are readily available for use.
3. Display in UI: The retrieved user data is then displayed in the user interface, ensuring a personalized and user-friendly experience. For instance, the user's name, email, and phone number may be displayed in input fields that are set to read-only mode, allowing users to view but not edit their information.
4. Clearing Data: AsyncStorage also plays a crucial role in allowing users to clear their order history. When a user confirms their intent to clear their order history, AsyncStorage is used to clear relevant data, such as the list of previously ordered books, from the local storage. This operation ensures that the app's data remains in sync with the user's actions.

In summary, AsyncStorage in our book rental application serves as a lightweight, local storage solution for managing user-related data and settings. It ensures that user information is persistently available across app sessions while allowing for user-initiated data clearing when necessary, providing a seamless and customizable user experience.

### 

### Implementation of Data Persistence

**SQLite Implementation**

**test.py**

A screen shot of a computer program

Description automatically generated

import sqlite3: This line imports the sqlite3 module, which provides a Python interface for working with SQLite databases.

db = sqlite3.connect('abcde.sqlite'): This line establishes a connection to an SQLite database named abcde.sqlite. If the database file doesn't exist, it will be created. If it already exists, the connection will be made to that existing database.

db.execute('DROP TABLE IF EXISTS booklist'): This SQL statement drops the table named booklist if it already exists. This is useful to ensure that you start with a clean slate. The IF EXISTS clause prevents an error from occurring if the table doesn't exist.

db.execute('''CREATE TABLE booklist( id integer PRIMARY KEY, name text NOT NULL, category text NOT NULL, image text NOT NULL, wishlist text NOT NULL, cart text NOT NULL, orders text NOT NULL, author text NOT NULL )'''): This SQL statement creates a new table named booklist with the following columns:

id: An integer column that serves as the primary key.

name: A text column that stores the name of the book.

category: A text column that stores the category of the book.

image: A text column that stores the path or URL to the image of the book.

wishlist: A text column that stores information related to the book being in a wishlist.

cart: A text column that stores information related to the book being in a shopping cart.

orders: A text column that stores information related to book orders.

author: A text column that stores the name of the author of the book.

cursor = db.cursor(): This line creates a cursor object, which allows you to execute SQL queries and fetch results from the database.

cursor.execute(''' ... '''): This line executes an SQL statement. The statement is enclosed in triple-quotes (''' ... ''') to allow multi-line strings in Python.

A black screen with orange text

Description automatically generated

There are 16 INSERT statements here. Each INSERT statement follows this structure:

INSERT INTO booklist(...): Indicates that data is being inserted into the booklist table.

(name,category,image,wishlist,cart,orders, author): Specifies the columns for which data will be provided.

VALUES(...): Indicates the values that will be inserted into the specified columns.

db.commit(): This line commits the changes made to the database. This means that the data added via the INSERT statements is permanently saved to the database.

db.close(): This line closes the connection to the database. It's important to close the connection after you're done with it to free up resources.

**webApi.py**

A screen shot of a computer program

Description automatically generated

import sqlite3: Imports the sqlite3 module for working with SQLite databases.

DB = 'android/app/src/main/assets/abcde.sqlite': Defines a constant DB which is a file path pointing to an SQLite database.

def get\_row\_as\_dict(row): Defines a function get\_row\_as\_dict that takes a database row as input and converts it into a dictionary. This function is used to easily convert database rows into JSON format.

A computer screen shot of text

Description automatically generated

GET Route: Retrieving Data (`/api/booklist`): This route retrieves a list of books from the SQLite database. It opens a database connection, executes an SQL query to select all records from the 'booklist' table, and fetches the results. The retrieved rows are then converted into dictionaries and returned as JSON using the `jsonify` function. This route demonstrates how data is read from the SQLite database and exposed as an API endpoint.

**AsyncStorage**

A computer screen shot of code

Description automatically generated

The async \_getList() function is responsible for retrieving data from AsyncStorage and updating the component's state with that data.

const keys = ['name', 'email', 'phone'];: Defines an array of keys ('name', 'email', 'phone') that you want to retrieve from AsyncStorage.

AsyncStorage.multiGet(keys, (err, stores) => {...});: This is an asynchronous call to AsyncStorage. It uses the multiGet method to retrieve the values corresponding to an array of keys.

stores.map((result, i, store) => {...});: This code block processes the result obtained from multiGet. It iterates over the results, where each result is an array with the first element being the key and the second element being the corresponding value. It then updates the newStates object with these key-value pairs.

this.setState(newStates);: Once all the key-value pairs are collected in newStates, this line sets the component's state with these values. This means that the component will be re-rendered with the new state, potentially reflecting changes in the UI based on the retrieved data.

The try...catch block is used for error handling. If any error occurs during the execution of the code inside the try block, it will be caught and handled in the catch block. In this case, any error will be logged to the console.

Overall, this function is designed to retrieve specific pieces of data (name, email, and phone) from AsyncStorage and update the component's state with these values, which are then likely used for displaying user information in the component.

A computer screen shot of a program code

Description automatically generated

The \_setItem function in the provided code is responsible for saving the name, email, and phone values to the device's storage using AsyncStorage. Here's how it works:

const {name, email, phone} = this.state; :This line extracts the name, email, and phone values from the component's state. These values are assumed to be controlled by input fields in the UI.

try { ... } catch (error) { ... }: This is a try-catch block, used for error handling. The code inside the try block is attempted, and if any errors occur, they are caught in the catch block.

await AsyncStorage.multiSet([['name', name], ['email', email], ['phone', phone]]): This line saves multiple key-value pairs using multiSet from AsyncStorage. It takes an array of arrays, where each inner array represents a key-value pair to be stored. In this case, it's saving three key-value pairs: ['name', name], ['email', email], and ['phone', phone]. The keys ('name', 'email', 'phone') are strings used to identify the values, and the corresponding values (name, email, phone) are the extracted values from the component's state.

If the operation is successful, the data will be stored in the device's storage.

If an error occurs during the storage operation, it will be caught by the catch block, and an error message will be logged to the console: console.log('## ERROR SAVING ITEM ##: ', error);.

In summary, the \_setItem function is used to save the name, email, and phone values to the device's storage. These values can be retrieved later using AsyncStorage when needed. This can be useful for persisting user data across sessions or app restarts.

**SQL queries**

1. List All Books (Retrieve operation):

* SELECT \* FROM booklist ORDER BY name;
* This query retrieves all rows from the booklist table and orders them by the name column in ascending order. It's used to fetch a list of all books.

2. Show a Book (Retrieve Operation):

* SELECT \* FROM booklist WHERE id=?;
* This query retrieves a single book from the booklist table based on its id value, which is provided as a parameter in the query.

3. Add a New Book (Create Operation):

* INSERT INTO booklist(name, category, image, wishlist, cart, orders, author) VALUES(?,?,?,?,?,?,?);
* This query inserts a new row into the booklist table with values provided as parameters. It adds a new book with attributes such as name, category, image, wishlist status, cart status, order status, and author.

4. Update a Book (Update Operation):

* UPDATE booklist SET name=?, category=?, image=?, wishlist=?, cart=?, orders=?, author=? WHERE id=?;
* This query updates an existing book's attributes in the booklist table based on its id. The new values for name, category, image, wishlist status, cart status, order status, and author are provided as parameters.

5. Delete a Book (Delete Operation):

* DELETE FROM booklist WHERE id=?;
* This query deletes a book from the booklist table based on its id.

6. Add to Wishlist (Update Operation):

* UPDATE booklist SET wishlist=? WHERE id=?;
* This query updates the wishlist status of a book in the booklist table based on its id. It sets the wishlist status to 'yes' for the specified book.

7. Add to Cart (Update Operation):

* UPDATE booklist SET cart=? WHERE id=?;
* This query updates the cart status of a book in the booklist table based on its id. It sets the cart status to 'yes' for the specified book.

8. Get Wishlist (Retrieve Operation):

* SELECT \* FROM booklist WHERE wishlist=? ORDER BY name;
* This query retrieves a list of books from the booklist table where the wishlist status is 'yes', and it orders them by the name column in ascending order.

9. Get Cart (Retrieve Operation):

* SELECT \* FROM booklist WHERE cart=? ORDER BY name;
* This query retrieves a list of books from the booklist table where the cart status is 'yes', and it orders them by the name column in ascending order.

10. Remove from Wishlist (Update Operation):

* UPDATE booklist SET wishlist=? WHERE id=?;
* This query updates the wishlist status of a book in the booklist table based on its id. It sets the wishlist status to 'no' for the specified book, effectively removing it from the wishlist.

11. Remove from Cart (Update Operation):

* UPDATE booklist SET cart=? WHERE id=?;
* This query updates the cart status of a book in the booklist table based on its id. It sets the cart status to 'no' for the specified book, effectively removing it from the cart.

12. Place Order (Update Operation):

* UPDATE booklist SET cart=?, orders=? WHERE cart=?;
* This query updates the cart status to 'no' and the orders status to 'yes' for all books in the cart. It marks the books in the cart as ordered.

13. Get Orders (Retrieve Operation):

* SELECT \* FROM booklist WHERE orders=? ORDER BY name;
* This query retrieves a list of books from the booklist table where the order status is 'yes', and it orders them by the name column in ascending order.

14. Clear Order History (Update Operation):

* UPDATE booklist SET orders=? WHERE orders=?;
* This query clears the order history by updating the orders status to 'no' for all books that were previously marked as ordered.

15. Search Books (Retrieve Operation):

* SELECT \* FROM booklist WHERE name LIKE ? ORDER BY name;
* This query searches for books in the booklist table based on a partial match of the name column. The search text is provided as a parameter with '%' wildcards added for flexibility, and the results are ordered by the name column in ascending order.

### CRUD Operations & Input Validation

1. Create operation:
   1. Add a New Book

A screenshot of a phone

Description automatically generated

This C operation of Add a New Book can be accessed by the admin through the Create screen. The admin needs to enter the book’s image, author, category, and name in order to create a new book.

1. Read /Retrieve operation:
   1. List All Books

A screenshot of a book

Description automatically generated

The R operation for List All Books will be available in the UserBookList and the BookList screen where the user or the admin can view all the books.

* 1. Show a Book

A screenshot of a phone

Description automatically generated

This R operation for Show a Book can be accessed through the View and the UserView screen. The user and admin can view the details of the book here.

* 1. Get Wishlist

A screenshot of a bookmark

Description automatically generated

This R operation of Get Wishlist can be accessed through the Wishlist screen where the user can find all the books that have been added to the wishlist.

* 1. Get Cart

A screenshot of a phone

Description automatically generated

This R operation of Get Cart can be accessed through the Cart screen where the user can find all the books that have been added to the cart.

* 1. Get Orders

A screenshot of a phone

Description automatically generated

This R operation of Get Orders will be accessed through the Order screen. The user can use the order information and the books that they have rented.

* 1. Search Books

A screenshot of a video game search

Description automatically generated

A screenshot of a phone

Description automatically generated

This R operation of Search Books will be accessed through the Search and UserBookList screen. If the user wants to search for books that have the particular books, he can enter the keyword at the search bar and press on the “search” button. It will return the books that have the particular keyword. For example: here, the user has entered the “hi” keyword to search for books that have the “hi” keyword, it will return the keyword that has the “hi” keyword. However, if the user entered a keyword that does not have any results, a message of “No results found” will appear.

1. Update operation
   1. Update a Book

A screenshot of a phone

Description automatically generated

This U operation of Update a Book can be accessed by the admin through the Edit Screen where the admin can update the book’s name, image, category, and author.

* 1. Add to Wishlist

A white background with black text

Description automatically generated

A white background with black text

Description automatically generated

This U operation of Add Wishlist can be accessed through the UserBookList screen, UserView screen, and the Search screen. It will allow the user to add a book into the wishlist. If the user has not added this book to the wishlist, a successful message of “Added to wishlist successfully” will pop up. If the user has added this book to the wishlist before, an error message of “Book is already in the wishlist” will pop up.

* 1. Add to Cart

A close-up of a text

Description automatically generated

A white background with black text

Description automatically generated

This U operation of Add to Cart can be accessed through the UserBookList screen, UserView screen, and the Search screen. It will allow the user to add a book into the cart. If the user has not added this book to the cart, a successful message of “Added to cart successfully” will pop up. If the user has added this book to the cart before, an error message of “Book is already in the cart” will pop up.

* 1. Remove from Wishlist

A white background with black text

Description automatically generated

A white background with black text

Description automatically generated

This U operation of Remove from Wishlist can be accessed through the Wishlist screen. The user will be asked whether he wants to confirm the removal of the particular book from the wishlist. If not, nothing happens. If yes, the book will be removed and the wishlist status of the book will be set to ‘no’.

* 1. Remove from Cart

A screenshot of a phone

Description automatically generated

A close-up of a person's face

Description automatically generated

This U operation of Remove from Cart can be accessed through the Cart screen. The user will be asked whether he wants to confirm the removal of the particular book from the cart. If not, nothing happens. If yes, the book will be removed and the cart status of the book will be set to ‘no’.

* 1. Place Order

A white background with black text

Description automatically generated

A white rectangular object with black text

Description automatically generated

This U operation of Place order can be accessed through the Order Screen. The user can click on the “Finish” button in the Order screen after filling up the order information. The user will be prompted to ask whether he wants to confirm the placement of the order. If not, nothing happens. If yes, the orders will be made by setting the order status of the book to ‘yes’ and the cart status of the book to ‘no’.

* 1. Clear Order History

A white background with black text

Description automatically generated

A white background with black text

Description automatically generated

The U operation of Clear Order History will be accessed through the Order screen where the user can remove the order history. Before he removes the order history, he will be prompted to confirm the removal of the order history. If not, nothing happens. If yes, the order history will be clear, and the order status of the book will be set to ‘no’.

1. Delete operation
   1. Delete a Book

A screenshot of a phone

Description automatically generated

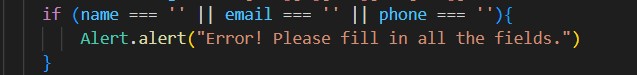
A close-up of a white background

Description automatically generated

This D operation of Delete a Book can be accessed by admin through the View screen where admin can delete a book. Before the admin deletes a book, he will receive a confirmation message asking whether he wants to delete the book. If not, nothing happens. If yes, a successful message of “Deleted successfully” will pop out.

**Input Validation**

1. Empty input or less input



A white background with black text

Description automatically generated

Our group has implemented the empty input validation in OrderScreen.js. In this code snippet, we can see that the user needs to make sure that he enters the information in the name, email, and phone number field. If the user does not complete filling all of the fields, there will be an alert message of “Error! Please fill in all the fields.” will pop up.

1. Input minimum and maximum length

A close up of a phone number

Description automatically generated

A computer screen with text

Description automatically generated

Our group has implemented this validation by only allowing the user to enter 10 digits of phone number in ContactUsScreen. This code snippet shows that if the user enters the length of the phone number is not equal to 10, an error message will be displayed. For example: if the user phone number is ‘098765432’, an error message will display since the phone number is not equal to 10 digits.

1. Input pattern (e.g., email format)

A white background with black text

Description automatically generated

A computer screen with text

Description automatically generated

We have also implemented the input pattern validation in the Order screen. In the code snippet, we can see that the user needs to make sure that he enters the correct format for the email. The correct format for the email is “something@something.something”. If the user inputs the email that does not follow this format, an error message of “Error! Incorrect email format” will be displayed.

# 

## CLOUD CONNECTIVITY

### Usage of Cloud Connectivity

**Web-Based Api**

A web-based API (Application Programming Interface) is a communication interface that enables software applications to interact over the internet using standard web protocols like HTTP or HTTPS. It allows one software component, often referred to as the client, to make requests to another component, typically a server hosting the API, to access data or perform specific functions. These APIs are accessed via URLs representing endpoints that correspond to particular resources or functionalities. Commonly, data is exchanged in formats like JSON or XML, and APIs may employ authentication, authorization, and response codes for secure and efficient communication. Web-based APIs are pivotal in enabling seamless integration between diverse applications, facilitating data exchange and functionality sharing across the web. Web-based APIs offer a multitude of advantages, including seamless interoperability across diverse software systems, scalability to handle varying workloads, cost-effective development through the use of pre-built functionalities, access to a vast array of third-party services and data, robust security measures, version control, and global accessibility. These APIs empower developers to rapidly create feature-rich applications while reducing data duplication and promoting real-time updates. They foster the growth of developer ecosystems and provide valuable insights through analytics, making them an indispensable component of contemporary software architecture.

By using Web-Based API, we had applied it into our book rental application. For example, clients can use the api to use our book rental application function such as Create book. Clients can send a request to our API's "create book" endpoint, providing the necessary data such as the book's title, author, ISBN, and other relevant details. Our API would then process this request, create a new book entry in your database, and return a response confirming the successful creation of the book or providing an error message if something goes wrong. This integration allows clients to interact with your book rental application programmatically, enabling them to perform various actions like adding, updating, or querying book information through the API, which can enhance the user experience and expand the functionality of your application.

**WebSocket API**  
 In our book rental application, the WebSocket API is employed to enable real-time communication between the mobile app and the server. This bi-directional communication mechanism facilitates data persistence and synchronization for various application features. Here's how WebSocket API is utilized:

1. **Real-Time Data Exchange**: WebSocket allows instant data exchange between the mobile app and the server. In our application, users can submit inquiries and feedback via the "Contact Us" feature. When a user submits a message, WebSocket ensures that the data is immediately sent to the server for processing.
2. **Server Handling**: On the server side, Flask and Flask-SocketIO are used to handle WebSocket connections. When a WebSocket connection is established from the mobile app, the server is notified of the connection, and it acknowledges this by printing a "connected" message. This establishes a persistent connection that remains open, enabling real-time interaction.
3. **Message Handling**: When a user submits a message through the "Contact Us" feature, the data is packaged into a JSON object and transmitted to the server via the WebSocket connection. The server processes the incoming message, extracting information such as the user's name, question, email, and contact number. We can customize this part to handle the data as needed.
4. **Broadcasting**: After processing the message, the server broadcasts a response message to all connected clients (users of the mobile app). This response includes a timestamp, user information, and the user's message. This broadcasting mechanism ensures that all users receive real-time updates, such as notifications of new inquiries or feedback.
5. **Success Feedback**: When the message is successfully processed on the server and broadcasted to clients, the mobile app displays a success modal to inform the user that their submission was successful. This enhances the user experience by providing immediate feedback.
6. **Validation and Error Handling**: The mobile app also includes validation and error handling to ensure that user-submitted data is accurate and complete. Validation checks include verifying email format and the format of the contact number. If there are validation errors, a modal is displayed, allowing users to correct their input.

In summary, WebSocket API facilitates real-time communication between the mobile app and the server, allowing for instant data persistence and synchronization. It ensures that user inquiries and feedback are promptly processed and that all connected clients receive updates in real time. This enhances the interactivity and responsiveness of the "Contact Us" feature in our book rental application.

### Implementation of Cloud Connectivity

Web Based Api

1. List All Books

A screenshot of a computer code

Description automatically generated

First define a route for the path with /api/booklist that will only respond to HTTP with GET requests. When a user sends a GET request, the function will be executed. Next will connect to the database and try to select all rows from the ‘booklist’ orders by the ‘name’ column. rows\_as\_dict=[] means will contain dictionaries representing the rows of data. At the last will return a JSON response to the user and convert the row\_as\_dict into a JSON response. The 200 status code represents that the request is successful.

A computer screen shot of text

Description automatically generated

This API is executed in the UserBookList screen and the BookList screen. Firstly, construct the URL for the API with the path /api/booklist. This sets the ‘isFetching’ state to true and initiates HTTP GET request to the URL which has been constructed in the first line. Then will start the response, if the response is not ok will display an alert message and throw an error with the text ‘Error’ followed by the HTTP status code. If the response is okay, then will set the ‘isFetching’ state to false and return to the JSON responses. If the JSON data parsing is successful, it will call the function that receives the parsed JSON data as books. console.log is used for debugging purposes and finally updates the component’s state with the fetched ‘books’ data. The last catch error is used to prepare for any errors occurring during the fetch operation, it will be caught in this ‘catch’ block.

1. Show a Book

A computer code with text

Description automatically generated

This line defines a route for the path /api/booklist/<int:book>, which expects an integer parameter book in the URL. It specifies that this route only responds to HTTP GET requests. Create a connection to an SQLite database and execute an SQL query to retrieve a single row from the booklist table based on the id provided in the URL. It uses a parameterized query where str(book) is passed as a parameter to avoid SQL injection vulnerabilities. The fetchone() method fetches the first matching row. After fetching, close the database connection and handle the response,If a matching row is found in the database , it transforms the fetched row into a dictionary format, likely using a function called get\_row\_as\_dict, and returns it as a JSON response using jsonify. The status code 200 indicates that the request was successful.

If no matching row is found, it returns a JSON response with None as its content but the response itself is still successful.

A computer screen with text on it

Description automatically generated

The first line constructs the URL for the API endpoint by combining the config.settings.serverPath with the path '/api/booklist/' and appending the this.state.bookID. The bookID appears to be a variable or state value representing the ID of the book to retrieve. The constructed URL is logged to the console for debugging purposes. Next, uses the fetch function to make an HTTP GET request to the URL.After the GET request is made, it chains a series of .then() blocks to handle the response: The first .then() checks if the response status is not okay using !response.ok. If the status is not okay, it displays an alert with the word 'Error:' followed by the HTTP status code using Alert.alert, and it throws an error to handle it in the next .catch() block.

If the response is okay, it proceeds to parse the JSON response using response.json(). If parsing the JSON data is successful, this code updates the component's state with the fetched books data. It appears to expect that the response will contain book details, and it sets the component's state to include this data. The ‘catch’ block is used to handle any errors that may occur duing the fetch operation. It logs the error to the console using ‘console.error’ for debugging purposes.

1. Add a new Book

A screenshot of a computer program

Description automatically generated

This code defines a route for handling HTTP POST requests at the path '/api/booklist.' It validates the incoming request for JSON data and extracts specific fields ('name,' 'category,' 'image,' 'wishlist,' 'cart,' 'orders,' and 'author') from the JSON object. The data is then inserted into an SQLite database, and the ID of the newly inserted record is retrieved. The changes are committed to the database, and a JSON response containing the new record's ID and the total changes made is returned with a 201 Created status code. Note that using a 404 status code for invalid or missing data might not be appropriate; a 400 Bad Request status code is more suitable.

A screen shot of a computer code

Description automatically generated

This line constructs the URL for the API endpoint by combining the config.settings.serverPath with the path '/api/booklist'. It forms the complete URL to the resource where the POST request will be sent. Next, using the fetch function to make anHTTP POST request to the URL.After the POST request is made, it chains a series of .then() blocks to handle the response: The first .then() logs the response object to the console for debugging purposes. It checks if the response status is not okay using !response.ok. If the status is not okay, it displays an alert with the word 'Error:' followed by the HTTP status code using Alert.alert, and it throws an error to handle it in the next .catch() block. If the response is okay, it proceeds to parse the JSON response using response.json(). If parsing the JSON data is successful, this code checks the respondJson object. If the 'affected' property in the response is greater than 0, it displays an alert with the message 'Added successfully.' Otherwise, it displays an alert with the message 'Error in SAVING.' The ‘catch()’ block is used to handle any errors that may occur during the POST request.

1. Update a Book

A screenshot of a computer program

Description automatically generated

First line defines a route for the path which expect an integer parameter ‘book’ in the URL and specifies this route only respond t HTTP PUT requests. Next checking the incoming request, if the request does not contain JSON data, aborts the request with a 400 Bad Request status code, if the JSON data does not contain ‘id’, it aborts with a 400 status code. If id in the JSON data does not match with the ‘book’ parameter provided in the URL, it also aborts the request with a 400 status code. Next, extracts the relevant data from the JSON request and prepares it for updating the database and stored in a tuple named ‘update\_member’. Connect the database and execute an SQL update statement to update an existing records in the ‘booklist’. table based on the id provided in the URL. It sets the field to the values fromm the update\_member tuple effectively modifying the book’s information. After that, commit the changes and create a JSON response containing the id of the updated book and the total number of changes made to the database. At the end close the database connection and returns a JSON response with the update books information and a status code of 201.

A screen shot of a computer code

Description automatically generated

This line constructs the URL for the API endpoint by combining the config.settings.serverPath with the path '/api/booklist/' and appending the this.state.booksId. The booksId variable appears to represent the ID of the book to retrieve. It logs the constructed URL to the console for debugging purposes. Next, use the fetch function to make an HTTP GET request to the URL.After the GET request is made, it chains a series of .then() blocks to handle the response: The first .then() checks if the response status is not okay using !response.ok. If the status is not okay, it displays an alert with the word 'Error:' followed by the HTTP status code using Alert.alert, and it throws an error to handle it in the next .catch() block.

If the response is okay, it proceeds to parse the JSON response using response.json().If parsing the JSON data is successful, this code updates the component's state with the fetched book details. It expects that the response will contain properties like 'name,' 'category,' 'image,' 'wishlist,' 'cart,' 'orders,' and 'author,' and it sets the component's state to include these properties with the corresponding values. Lastly, using catch() block to handle any errors that may occur during the fetch operations.

1. Delete a Book

A computer screen with text and images

Description automatically generated

The first line define a route for the path where int book is a placeholder for an integer parameter named book. It specifies that this route only response to HTTP DELETE requests. Next fech if the request does not contain JSON data. If it doesn't, it aborts the request with a 400 Bad Request status code. If the JSON data does not contain an 'id' field. If 'id' is missing, it aborts the request with a 400 status code. If the 'id' in the JSON data does not match the 'book' parameter provided in the URL. If they don't match, it also aborts the request with a 400 status code. Connect the database and execute an SQL DELETE statement to remove a book record from the booklist table based on the id provided in the URL an uses a parameterised query to ensure proper handling of the id values. After that, commit the changes, and creates a JSON response containing the 'id' of the deleted book (taken from the URL) and the total number of changes made to the database. Finally, close the database connection and return a JSON response with the deleted book’s information and a status code of 201.

A screen shot of a computer screen

Description automatically generated

This first line constructs the URL for the API and appending the this.state.bookID. The bookID variable appears to represent the ID of the book to delete. Next, using the fetch function to make an HTTP DELETE request to the URL.It specifies the request method as 'DELETE'. It includes headers to indicate that the client can accept JSON responses (Accept: 'application/json') and that the request body contains JSON data ('Content-Type': 'application/json'). The request body is constructed using JSON.stringify, including an object with an 'id' property set to this.state.bookID. This object is provided to indicate which book should be deleted. After the DELETE request is made, the then block to handle the response. The first .then() checks if the response status is not okay using !response.ok. If the status is not okay, it displays an alert with the word 'Error:' followed by the HTTP status code using Alert.alert, and it throws an error to handle it in the next .catch() block. If the response is okay, it proceeds to parse the JSON response using response.json().Next is response handling, If parsing the JSON data is successful, this code checks the 'affected' property in the response JSON object. If 'affected' is equal to 0, it displays an error alert ('Error in DELETING') to indicate that the deletion operation did not affect any records. If 'affected' is not equal to 0, it displays an alert indicating successful deletion ('Deleted successfully'). At the last, the catch block is used to handle any errors that may occur during the DELETE request.

1. Add to Wishlist

A screenshot of a computer code

Description automatically generated

The first line defines a route for the path /api/booklist/add\_to\_wishlist/<int book>’ which is an integer parameter ‘book’ in URL and specifies the HTTP PUT request. Next, if the request does not contains JSON data. It aborts the request with a 400 Bad request status code to ensure that the user sends valid JSON data. Next, it will execute an SQL query to retrieve a book from the ‘booklist’ table by its ‘id’ and use ‘str(book)’ to convert the ‘book’ parameter from the URL into a strings then fetches the first matching row. If the book exists in the database, it checks the value of the wishlist field (assuming it's at index 4) in the database row. If the value is 'yes,' it means the book is already in the wishlist. If the book is not in the wishlist, this code updates the wishlist field for that book to 'yes' and commits the change to the database. At the end will close the database connection and returns a JSON response with a message and a 201 status which represent Created. If the book is already in the wishlist will returns a 400 Bad Request status code and if not found will returns 404.

A screen shot of a computer code

Description automatically generated

It constructs the URL for the PUT request and appending the ‘book.id’ to specify which books should be added to the wishlist. Handels the responses by parsing it as JSON and either display an alert with a message form the response or logs an error message.

1. Add to Cart

A screenshot of a computer code

Description automatically generated

First line defines a route for the path which expected an integer parameter ‘book’ in the URL with the HTTP PUT requests. If request does not contains JSON data will aborts the request with a 400 Bad Request status code. Connect the database and check if the book exists in the database, checks the value of the ‘cart’ in the database row. If the value is ‘yes’ means the book is already in the cart. If no, will update the ‘cart’ for the book and commit changes to the database. Finally will close the database and return a JSON message with a 201 Created status code. if the book is already in cart will return a 400 and if book is not found will return 404 which is Not Found Status code.

A screen shot of a computer code

Description automatically generated

First line construct the URL for the PUT request by combining with the path api/booklist/add\_to\_cart/ and appending the ‘book.id’ to specify which book should be added into the cart. Next, using the fetch function to make HTTP PUT requests to the URL. After that, the first then parses the response body as JSON using response.JSON logs the entire response object to the console for debugging purposes and the second then takes the parsed JSON data as ‘data’. If ‘data’ contains a ‘message’ property, it display an alert with the message and if not, it will logs the response. The last section is used to handle any errors that may occur during the request.

1. Get Wishlist

A computer screen shot of a computer code

Description automatically generated

First line defines the path and the HTTP GET request, connect to the database and execute an SQL query to retrieve all rows from the ‘booklist’ table where the ‘wishlist’ field set to ‘yes’ and ordered the result by the ‘name’ column. After fetching the result and close the connection of the database. Next , converting the row fetched from the database into a list of dictionaries. Lastly, the line returns a JSON request to the client and uses the ‘jsonify’ function to convert the list of dictionaries into a JSON request which is in 200 status code.

A computer screen shot of a program

Description automatically generated

Firstly, construct the URL for the API with the path and sets the isFetching state to true. Next, use the fetch function to make an HTTP GET request to the URL just now.After that, the GET request is made, will start handling the response by 2 then. First then is used to checks if the response if not okay, display an alert message and throws an error to handle it in the next ‘.catch()’ block. It the response is okay, set the isFecthing status to ‘false’ and parses the JSON request using response.json(). If parsing the JSON data is successful, this code logs the parsed JSON data to the console for debugging and updates the component's state with the fetched books data. The .catch() block is used to handle any errors that may occur during the fetch operation. It logs the error to the console for debugging.

1. Get Cart

A computer screen with text

Description automatically generated

This line defines a route for the path that will only respond to a HTTP GET requests. Create a connection to an SQLite database and create a cursor to interact with the database. Execute an SQL query to retrieve all rows from the ‘booklist’ table where the ‘cart’ field is set to ‘yes’. It orders the result by the ‘name’ column. The query uses a parameterized query where yes is passed as a parameter to avoid SQL injection vulnerabilities. It fetches all matching rows into the ‘row’ variables. After that, close the database and converting the rows fetched from the database into a list of dictionaries. It likely uses a function get\_row\_as\_dict to transform each row into a dictionary format. The resulting list, rows\_as\_dict, will contain dictionaries representing the rows of data. Lastly, return a JSON responses to the users. Use the jsonify function to convert the list of dictionaries into a JSON response and 200 status code indicates that the request was successful.

A computer screen with text and images

Description automatically generated

The first line constructs the URL for the API endpoint by combining theconfig.settings.serverPath with the path /api/booklist/cart. Next, the component state variable ‘isFetching’ set to true. Use the fetch function to make an HTTP GET request to the URL. After the GET request is made, the ‘then()’ blocks to handle the response.The first .then() block logs the response object to the console for debugging purposes. It then checks if the response status is not okay using !response.ok. If the status is not okay, it displays an alert with the word 'Error:' followed by the HTTP status code using Alert.alert, and it throws an error to handle it in the next .catch() block. If the response is okay (status code 200), it sets the isFetching state variable to false to indicate that data fetching is complete, and it proceeds to parse the JSON response using response.json(). Next, ff parsing the JSON data is successful, this code logs the parsed books data to the console for debugging purposes. It then sets the component's state variable books to the fetched list of books. This update allows the component to render the fetched data in the UI. Lastly, the catch block is used to handle any errors that may occur during the GET request or response parsing.

1. Remove from Wishlist

A screenshot of a computer program

Description automatically generated

The first line define the route for the path where int book is a placeholder for an integer parameter named book. It specify that this route only responds to HTTP PUT request. Next, If the request does not contain JSON data, it aborts the request with a 400 Bad Request status code, indicating that the request is invalid. Connect to SQLite database and execute the UPDATE statement to remove a book from the wishlist. It sets the 'wishlist' column to 'no' for the book with the specified 'id' (provided in the URL). Commit the changes and creates a JSON response containing a simple message indicating that the book has been successfully removed from the wishlist. It returns a status code of 200 OK to indicate a successful request and close the database connection.

A screen shot of a computer program

Description automatically generated

This line construct the URL for the API endpoint and appending the book id to target an APIendpoint for removing a book from the user’s wishlist. Hence, use the fetch function to make an HTTP PUT request to the URL. It specifies the request method as 'PUT'. It includes headers with the 'Content-Type' set to 'application/json' to indicate that the request body contains JSON data. The request body is constructed using JSON.stringify(book), where book appears to be an object representing the book to be removed from the wishlist. After PUT request, then block to handle the response.If parsing the JSON data is successful, this code logs the entire response object to the console for debugging purposes. It then checks if the response JSON contains a 'message' property using data && data.message. If a 'message' exists, it displays an alert with the message using Alert.alert(data.message) and calls a function \_query(). The \_query() function is presumably used to refresh the wishlist data after removing the book. If the response does not contain a 'message' property, it logs a message to the console indicating that the response did not contain a message. Lastly, the catch block is used to handle any errors that may occurs during the PUT request or response parsing.

1. Remove from Cart

A screen shot of a computer code

Description automatically generated

This line defines a route for the path where the int book is a placeholder for an integer parameter named books. It specifies that this route only responded to HTTP PUT requests. Next, if the request does not contain JSON data, it aborts the request with a 400 Bad Request status code, indicating that the request is invalid. Connect the database and execute the SQL UPDATE statement to remove a book from the user’s cart. It set the ‘cart’ column to ‘no’ for the book with the specified id. Commit the changes and creates a JSON response containing message and return a status code of 200.

A screen shot of a computer screen

Description automatically generated

Constructing the URL: It constructs the URL for the API endpoint by combining the config.settings.serverPath with the path /api/booklist/remove\_from\_cart/ and appends the book.id. This URL targets the API endpoint designed for removing a book from the user's cart. Next,making the PUT Request. It uses the fetch function to make an HTTP PUT request to the URL constructed in the previous step. It specifies the request method as 'PUT', includes headers with 'Content-Type' set to 'application/json' to indicate JSON data in the request body, and sends a JSON representation of the book object in the request body. Next, handling the response: After the PUT request is made, it handles the response in a series of .then() blocks. It expects a JSON response, so it parses it with response.json(). If successful, it logs the entire response object, checks if the response contains a 'message,' and displays that message using Alert.alert. It also calls the \_query() function, presumably to refresh cart data if the book removal was successful. If the response doesn't contain a message, it logs that information. Lastly for the Error Handling, It includes a .catch() block to handle any errors that may occur during the request or response handling, logging the error to the console.

1. Place Order

A computer screen with text on it

Description automatically generated

This line defines a route for the path that specifically responds to HTTP PUT requests. Next, checking the request, if there is no JSON data in the request, abort the request with a 400 status code. Connect the database and UPDATE the statement in the user cart to the ordered state. It set the cart column to no and the orders column to yes for items that were previously in the cart ‘yes’. After that comimit, return a JSON response containing a simple message indicating that the order has been successfully placed and return a status code of 200.

A computer screen shot of a program code

Description automatically generated

First, it constructs the URL for the API endpoint by combining config.settings.serverPath with the path /api/booklist/place\_order. This URL targets the API endpoint designed for placing an order. Next, uses the fetch function to make an HTTP PUT request to the URL constructed in the previous step. It specifies the request method as 'PUT', includes headers with 'Content-Type' set to 'application/json' to indicate JSON data in the request body, and sends a JSON representation of this.state.books in the request body. After the PUT request is made, it handles the response in a series of .then() blocks. It expects a JSON response, so it parses it with response.json(). If successful, it logs the entire response object, checks if the response contains a 'message,' and displays that message using Alert.alert. It also navigates to the 'Home' screen and resets the component's state variables if the response contains a message. If the response doesn't contain a message, it logs that information. Lastly, It includes a .catch() block to handle any errors that may occur during the request or response handling, logging the error to the console.

1. Get Order

A computer screen with text

Description automatically generated

This line defines a route for the path /api/booklist/orders, specifically responding to HTTP GET requests.It establishes a connection to an SQLite database specified by the DB variable and creates a cursor to interact with the database. Cursors allow you to execute SQL queries on the database. Execute a SELECT statement to retrieve records from the booklist table where the 'orders' column is set to 'yes'. It orders the results by the 'name' column in ascending order. Before responding, it iterates through the retrieved rows and converts each row into a dictionary format using a function named get\_row\_as\_dict. These dictionaries are appended to the rows\_as\_dict list. and finally, it responds with a JSON representation of the list of ordered books (in dictionary format). The status code 200 OK indicates a successful request.

A computer screen with text and images

Description automatically generated

First, it constructs the URL for the API endpoint by combining config.settings.serverPath with the path /api/booklist/orders. This URL targets the API endpoint designed for fetching ordered books. Before making the request, it sets the component's isFetching state to true. This is often used to indicate that data is currently being fetched and can be used to show loading indicators in the user interface. Next, It uses the fetch function to make an HTTP GET request to the URL constructed in the previous step. It does not specify the request method because GET is the default method for fetch.

After the GET request is made, it handles the response in a series of .then() blocks. It first checks if the response is OK (status code 200) using response.ok. If the response is not OK, it displays an error message using Alert.alert and throws an error to trigger the .catch() block. If the response is OK, it continues by parsing the response as JSON using response.json(). Once the JSON response is parsed successfully, it logs the retrieved books to the console and sets the component's books state to the fetched data. This likely updates the component's UI with the retrieved book data. Lastly It includes a .catch() block to handle any errors that may occur during the request or response handling. If an error occurs, it logs the error to the console. After the request and response handling are complete, it sets the isFetching state back to false, indicating that data fetching has finished.

1. Clear Order History

A computer screen with text on it

Description automatically generated

The line defines a route for the path and responding to HTTP PUT requests. if the incoming request contains JSON data. If it doesn't, it aborts the request with a 400 Bad Request status code. Establishes a connection to an SQLite database and executes an SQL UPDATE statement to modify records in the 'booklist' table. It sets the 'orders' column to 'no' for all records where the 'orders' column is currently 'yes'. commits the changes to the database and lastly, It responds with a JSON object containing a message indicating that the order history has been cleared successfully. The status code 200 OK indicates a successful request.

A computer screen with text and images

Description automatically generated

First, It constructs the URL for the API endpoint by combining config.settings.serverPath with the path /api/booklist/clear\_history. This URL targets the API endpoint designed for clearing order history. Next, It uses the fetch function to make an HTTP PUT request to the URL constructed in the previous step. It specifies the request method as 'PUT', includes headers with 'Content-Type' set to 'application/json' to indicate JSON data in the request body, and sends a JSON representation of this.state.books in the request body. After the PUT request is made, it handles the response in a series of .then() blocks. It expects a JSON response, so it parses it with response.json(). If successful, it logs the entire response object, checks if the response contains a 'message,' and displays that message using Alert.alert. It also logs a success message to the console and calls the \_query() function to presumably refresh data. If the response doesn't contain a message, it logs that information. Last, it includes a .catch() block to handle any errors that may occur during the request or response handling, logging the error to the console.

1. Search Books

A computer screen with text

Description automatically generated

This line defines a route for the path /api/booklist/search/<string:text> and specifies that it responds to HTTP GET requests. The <string:text> part in the URL is a dynamic parameter, allowing you to pass a text query as part of the URL.It establishes a connection to an SQLite database specified by the DB variable and creates a cursor to interact with the database. Cursors are used to execute SQL queries on the database.This code executes an SQL SELECT statement to search for records in the 'booklist' table where the 'name' column contains the provided text parameter. The % signs are used as wildcards for pattern matching. The results are ordered by the 'name' column. After executing the SQL query, this line fetches all the rows that match the query criteria. Finally, this code closes the database connection to free up resources since the data retrieval is complete. It converts the fetched database rows into a list of dictionaries, where each dictionary represents a book. Then, it responds with a JSON representation of these dictionaries using Flask's jsonify function. The status code 200 OK indicates a successful request. This block of code allows you to run the Flask application from the command line. It parses command-line arguments to specify the port on which the server should listen. The app.run call starts the Flask application on the specified host and port.

A computer screen shot of a program code

Description automatically generated

First, It constructs the URL for the API endpoint by combining config.settings.serverPath with the path /api/booklist/search/ and appending the text parameter to the end of the URL. This URL targets the API endpoint designed for searching books by name. Next, It uses the fetch function to make an HTTP GET request to the constructed URL. Since it's a GET request, it doesn't need to specify the request method; GET is the default method for fetch. After the GET request is made, it handles the response in a series of .then() blocks. First, it checks if the response is OK (status code 200) using response.ok. If the response is not OK, it displays an error message using Alert.alert and throws an error to trigger the .catch() block. If the response is OK, it continues by parsing the response as JSON using response.json(). Once the JSON response is successfully parsed, it logs the retrieved books to the console and sets the component's books state to the fetched data. This likely updates the component's UI with the search results. Lastly, It includes a .catch() block to handle any errors that may occur during the request or response handling. If an error occurs, it logs the error to the console.

**WebSocket API for Contact Us**

This is a Python Flask application that uses Flask-SocketIO for handling WebSocket connections. The application consists of a web page with a contact form. When the form is submitted, the data is sent to the server using a WebSocket connection, processed, and then sent back

A screen shot of a computer program

Description automatically generated

A screen shot of a computer code

Description automatically generated

In our book rental application, the WebSocket API is utilized to establish a real-time connection between the application and a cloud server for sending and receiving data. Here's how WebSocket API is used to establish and manage this connection:

1. **Initialization**: We start by creating a Flask application (**apps**) and configure it with a secret key. Then, we initialize a **SocketIO** instance (**socketio**) associated with the Flask app. This prepares the server to handle WebSocket connections.
2. **Routing**: We define a route for the root URL ('/'). This is typically used to serve an initial HTML page or to handle other non-WebSocket HTTP requests. In our case, we return an HTML template ('index.html').
3. **WebSocket Events**: WebSocket communication is event-driven. We define several event handlers using **@socketio.on** decorators. These events correspond to different phases of the WebSocket connection and data exchange.
   * **connect** Event: When a client establishes a WebSocket connection, the **handle\_connect\_chat** function is executed. It logs the message 'connected,' indicating that a client has successfully connected to the WebSocket server.
   * **web\_client\_connected** Event: This event is triggered when a web client (the mobile app in our case) connects to the WebSocket server. The function **handle\_client\_connected\_chat** logs the connection status, providing feedback about the client's connectivity.
   * **contact\_us\_message** Event: This event is used for handling messages sent by the mobile app's users via the "Contact Us" feature. When a message is received, the **handle\_contact\_us\_message** function extracts the user's name, question, email, and contact number from the incoming data. We can customize this part to handle the data as needed. For demonstration purposes, the received message is emitted back to all connected clients (broadcasted).
4. **Data Processing**: Inside the **handle\_contact\_us\_message** function, we process the received data. In this case, we extract relevant information and create a response JSON object containing a timestamp, user's name, question, email, and contact number.
5. **Broadcasting**: After processing the message, we use the **emit** function to broadcast the response data to all connected clients. This ensures that every client receives the message in real time, enhancing the interactivity of our "Contact Us" feature.
6. **Server Initialization**: Finally, we check if the script is the main program, and if so, we run the Flask app with SocketIO support using **socketio.run(apps)**.

In summary, the WebSocket API is crucial for establishing a real-time connection between our book rental application and a cloud server. It allows clients to connect, send messages (in this case, user inquiries and feedback), and receive responses in real time. This bi-directional communication enhances user engagement and provides immediate feedback, improving the overall user experience.

**"Contact Us" form**

This is an HTML file that serves as a web page for a "Contact Us" form. It includes some basic HTML structure along with JavaScript to handle WebSocket communication. The HTML and JavaScript code is for creating a web-based "Contact Us" page that interacts with a Flask server using WebSocket API technology. This code is responsible for displaying contact form responses in real-time on the web page as they are received from clients.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generatedA screen shot of a computer

Description automatically generatedA screen shot of a computer program

Description automatically generated

Let's break down the usage of the codes above:

1. **HTML Structure**:
   * The HTML file defines the structure of the "Contact Us" web page.
   * It includes a title, a header displaying "Contact Us," and a table for displaying contact form responses.
2. **Socket.IO Library Inclusion**:
   * The code includes the Socket.IO client library using a **<script>** tag. This library allows the web page to establish WebSocket connections and communicate with the server in real-time.
3. **Table for Displaying Responses**:
   * A table is defined within the HTML structure to display contact form responses.
   * The table has a header row with column titles such as "Date & Time," "Name," "Question/Feedback," "Email," and "Contact Number."
4. **Real-Time Display of Responses**:
   * When a response is received from the server, the JavaScript code parses the JSON data and dynamically adds a new row to the table with the response details. This enables real-time display of contact form responses as they are received.
5. **Styling**:
   * The code includes CSS styles for styling the HTML elements, ensuring that the page and the table are visually appealing.

For the JavaScript code, it shows how the Contact Us form uses the WebSocket API to display contact form responses in real-time on the web page as they are received from clients. Here's a step-by-step explanation of how this code achieves real-time updates:

1. **WebSocket Initialization**:
   * The code initializes a WebSocket connection by creating a **socket** object using **io.connect**. It specifies the WebSocket server's URL, which includes the document's domain and port, as well as the namespace ("/chat") to which it connects. This matches the namespace defined on the server side.
2. **'connect' Event Handler**:
   * Upon successfully connecting to the WebSocket server, the code executes the **connect** event handler function. This function emits a message to the server using **socket.emit**. It sends an object **{ connected: true }** to notify the server that the web client is connected. This step helps the server keep track of client connections.
3. **'contact\_us\_response' Event Handler**:
   * The code defines an event handler for the 'contact\_us\_response' event. This event is triggered when the server sends a response containing contact form data.
4. **Response Processing**:
   * Inside the 'contact\_us\_response' event handler, the code receives the **data** parameter, which contains the JSON data sent by the server.
   * It parses the JSON data into a JavaScript object (**response**) using **JSON.parse(data)**.
5. **Dynamic Table Update**:
   * To display the received response in real-time, the code dynamically updates the HTML table.
   * It selects the **<tbody>** element with the ID 'responseTableBody' using **document.getElementById('responseTableBody')**.
6. **Creating a New Table Row**:
   * The code creates a new table row (**<tr>**) by inserting a new row into the selected **<tbody>** element using **insertRow(0)**. The **0** index ensures that new rows are added at the top of the table, making the most recent responses appear first.
7. **Adding Table Cells**:
   * Within the new table row, the code inserts cells (**<td>**) to hold each piece of response data. It inserts cells for the following data: timestamp, name, question, email, and contact number.
8. **Populating Cells with Data**:
   * The code populates each cell with data from the received response. It assigns the corresponding values from the **response** object to the cell's **textContent**.
9. **Console Logging**:
   * For debugging purposes, the code logs messages to the browser's console to indicate the connection status and when responses are received from the server.

In summary, this JavaScript code establishes a WebSocket connection with the server and listens for 'contact\_us\_response' events. When a response is received, it dynamically adds a new row to the HTML table at the top, displaying the contact form response details. This process ensures that responses are displayed in real-time on the web page as they are received from clients, creating an interactive and live feedback display for the Contact Us form.

**ContactUsScreen**

The code segment below demonstrates how the Contact Us form uses the WebSocket API for cloud connectivity, allowing users to contact the application's administrators in a seamless and interactive manner.

const handleSubmit = () => {

// Initialize the errors object

const validationErrors = {};

// Validate that all fields are filled

if (!name || !question || !email || !contactNumber) {

validationErrors.allFields = 'All fields are required';

}

// Validate email format

const emailPattern = /^[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$/i;

if (!emailPattern.test(email)) {

validationErrors.email = 'Invalid email address';

}

// Validate phone number format (assuming a simple format with 10 digits)

const phoneNumberPattern = /^\d{10}$/;

if (!phoneNumberPattern.test(contactNumber)) {

validationErrors.contactNumber = 'Invalid phone number (10 digits required)';

}

// Update the errors state with the validation errors

setErrors(validationErrors);

// If there are validation errors, show the error modal

if (Object.keys(validationErrors).length > 0) {

setModalVisible(true);

return;

}

// If validation passes, send the data to the server

const socket = io('http://10.0.2.2:5000/chat'); // Replace with your server URL

const contactData = {

name,

question,

email,

contactNumber,

};

socket.emit('contact\_us\_message', contactData);

// Show the success modal

setSuccessModalVisible(true);

};

const resetForm = () => {

setName('');

setQuestion('');

setEmail('');

setContactNumber('');

setErrors({});

};

const handleModalClose = () => {

setModalVisible(false);

};

const handleSuccessModalClose = () => {

setSuccessModalVisible(false);

// Only reset the form when the success modal is closed and there are no validation errors

if (!Object.keys(errors).length) {

resetForm(); // Call the resetForm function

}

};

Here's how it works:

1. **Form Validation**:
   * Before sending any data to the server, the form data is first validated to ensure that it meets the required criteria.
   * Validation errors are stored in the **validationErrors** object. It checks if all fields (name, question, email, and contactNumber) are filled and whether the email and contact number formats are valid.
2. **Error Handling**:
   * If there are validation errors (**Object.keys(validationErrors).length > 0**), an error modal is displayed to the user via **setModalVisible(true)**. This provides immediate feedback to the user about any issues with their input.
3. **WebSocket Initialization**:
   * If validation passes and there are no errors, a WebSocket connection is established. It connects to the server using the URL '<http://10.0.2.2:5000/chat>'. This URL should match the WebSocket server's location and namespace.
4. **Data Preparation**:
   * The contact form data (name, question, email, and contactNumber) is collected into a JavaScript object named **contactData**. This object will be sent to the server via WebSocket.
5. **WebSocket Data Transmission**:
   * The **socket.emit** method is used to send the **contactData** object to the server. It emits an event named 'contact\_us\_message' along with the data. This event corresponds to the server's 'contact\_us\_message' event handler.
6. **Success Modal Display**:
   * After sending the data to the server, a success modal is displayed to the user using **setSuccessModalVisible(true)**. This informs the user that their message has been sent successfully.
7. **Form Reset (on Success)**:
   * If the user closes the success modal and there are no validation errors (**!Object.keys(errors).length**), the form fields are reset to their initial empty state. The **resetForm** function is called to achieve this.
8. **Modal Handling**:
   * Two functions, **handleModalClose** and **handleSuccessModalClose**, handle the closing of the error modal and success modal, respectively. These functions are called when the user interacts with the modals' close buttons.

In summary, the Contact Us form uses the WebSocket API to connect to the server and send contact form data for cloud connectivity. It performs client-side validation to ensure data integrity and provides immediate feedback to the user. When data is successfully sent, it notifies the user through a success modal and optionally resets the form. This approach allows users to contact the application's administrators and receive real-time feedback, enhancing the user experience.

**END OF DOCUMENT**