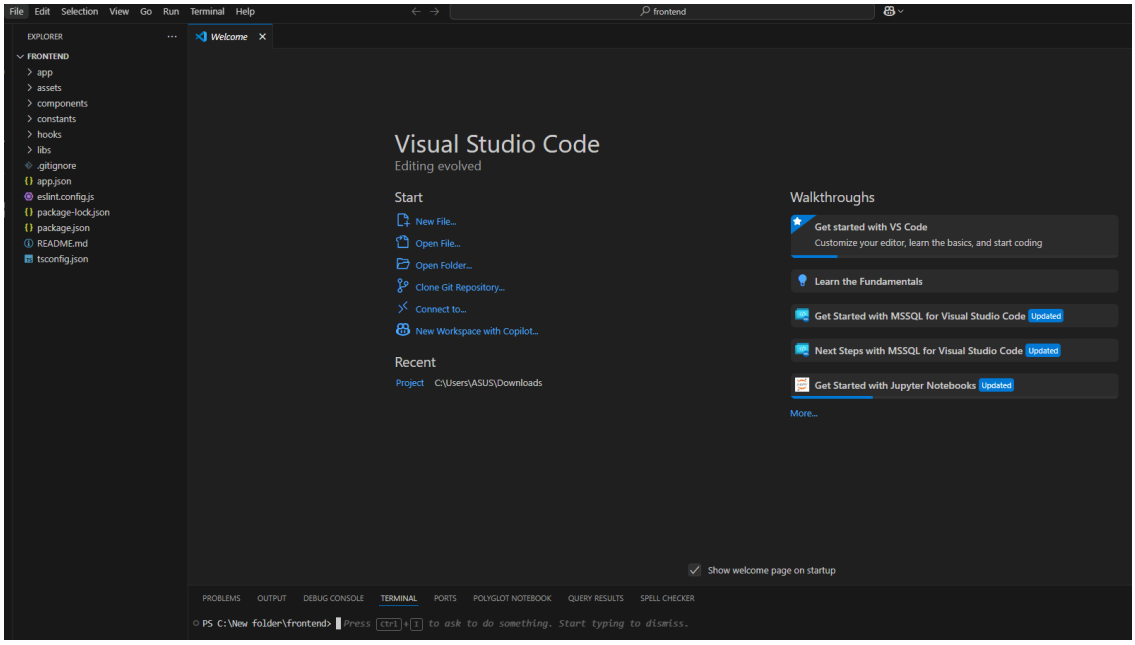
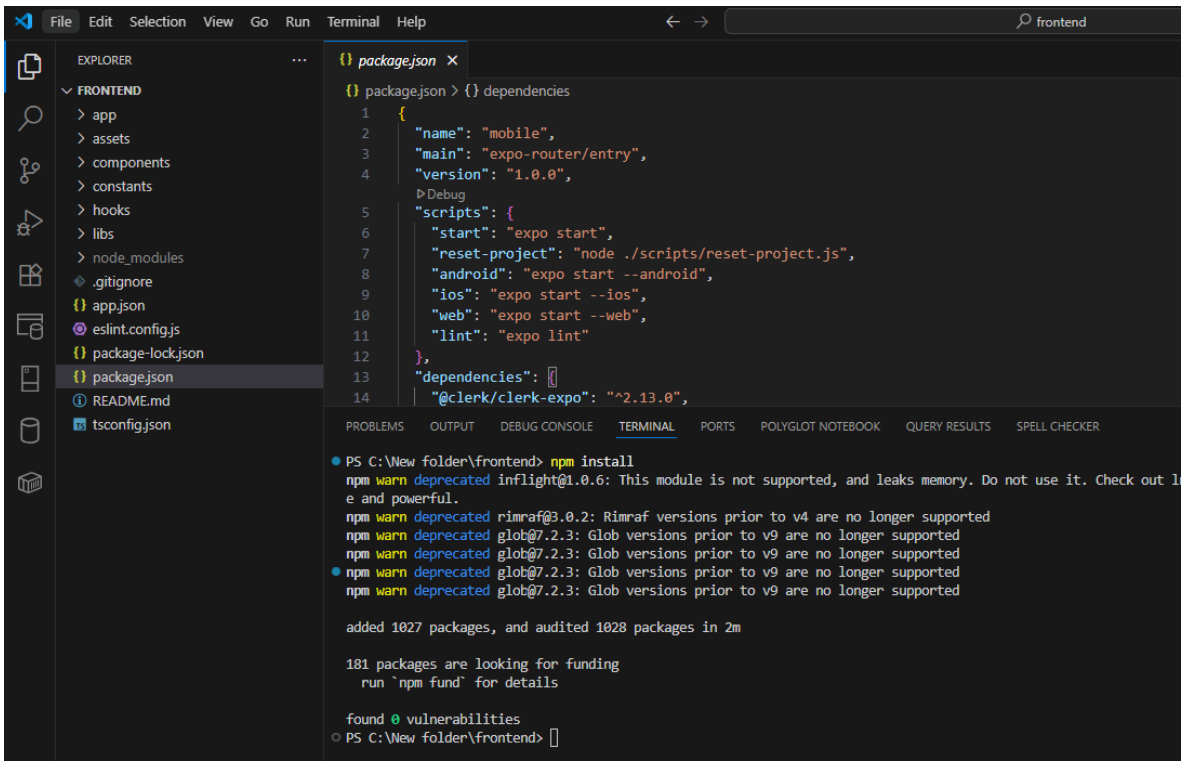
How to setup the Front-End React Native Application with Docker

Setting up the Front-End project

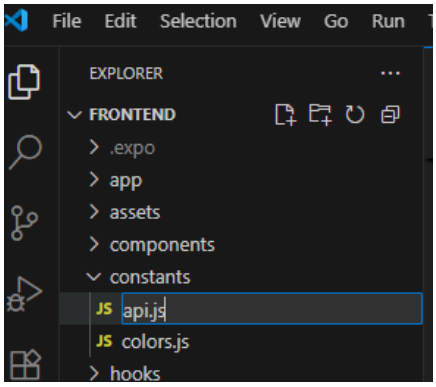
Open up the React native project using an IDE, using Visual Studio Code in example.



In the terminal, run npm install, ensure that the terminal is pointing to the directory in the project folder. This will install all of the dependencies listed in the package.json file.



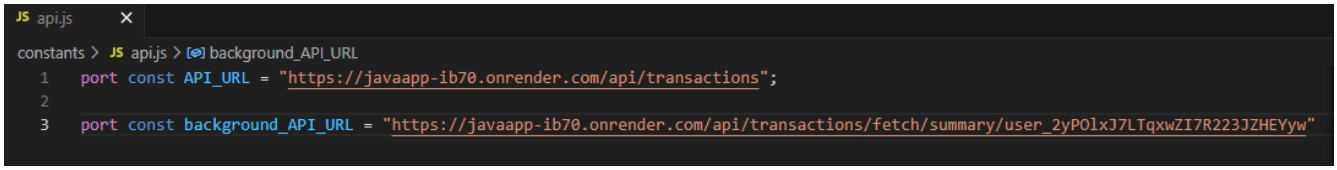
Then go to constants folder and create a new file called api.js. This contains the environment variables for the project.



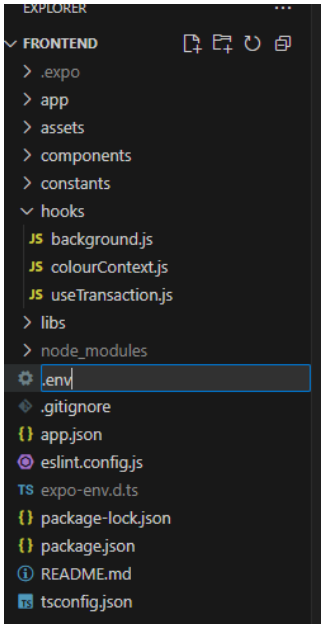
Paste the following lines into the file:

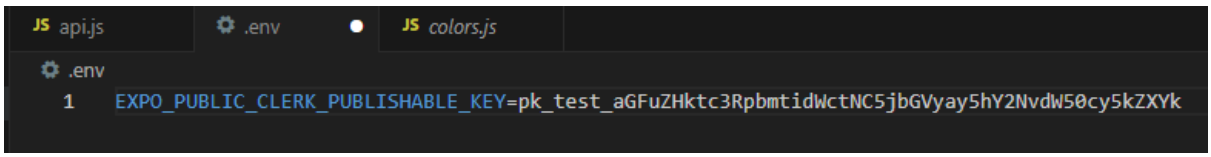
export const API_URL = "https://javaapp-ib70.onrender.com/api/transactions";

export const background_API_URL = "https://javaapp-ib70.onrender.com/api/transactions/fetch/summary/user_2yPOlxJ7LTqxwZI7R223JZHEYyw"



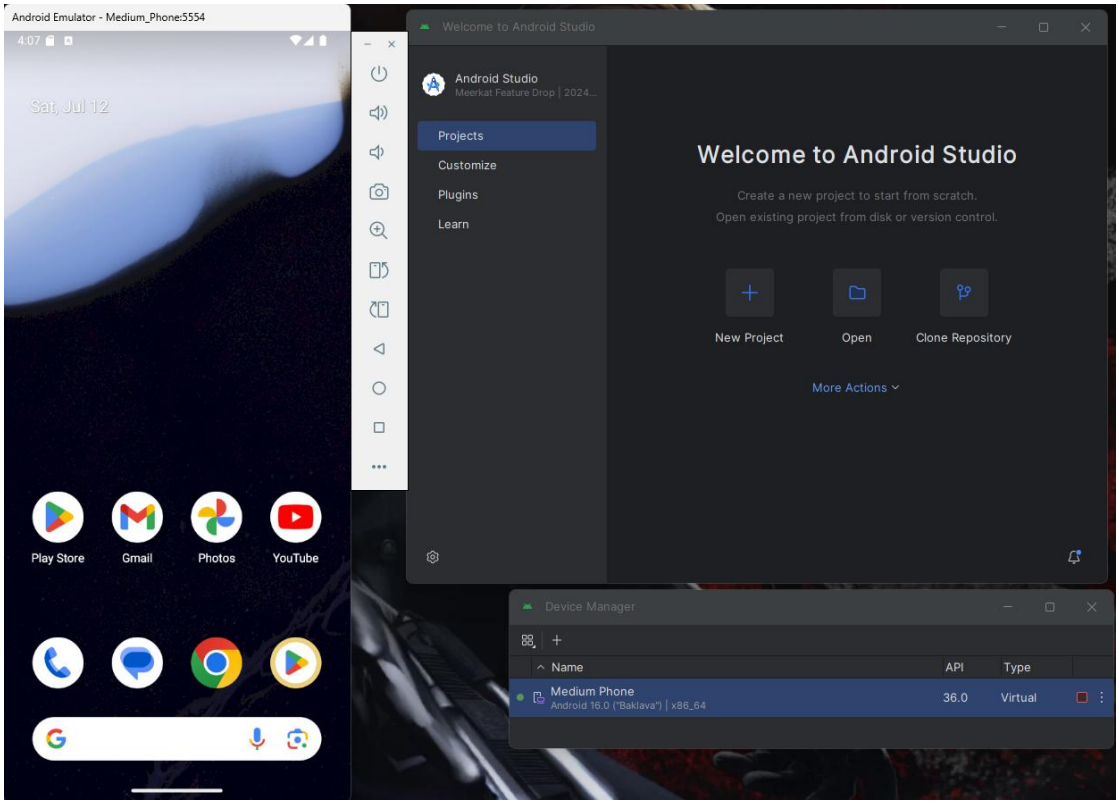
Also, create a new file in the project directory called .env. This also contains an environment variable. Paste the following line into the file:
EXPO_PUBLIC_CLERK_PUBLISHABLE_KEY=pk_test_aGFuZHktc3RpbmtidWctNC5jbGVyay5hY2NvdW50cy5kZXYk



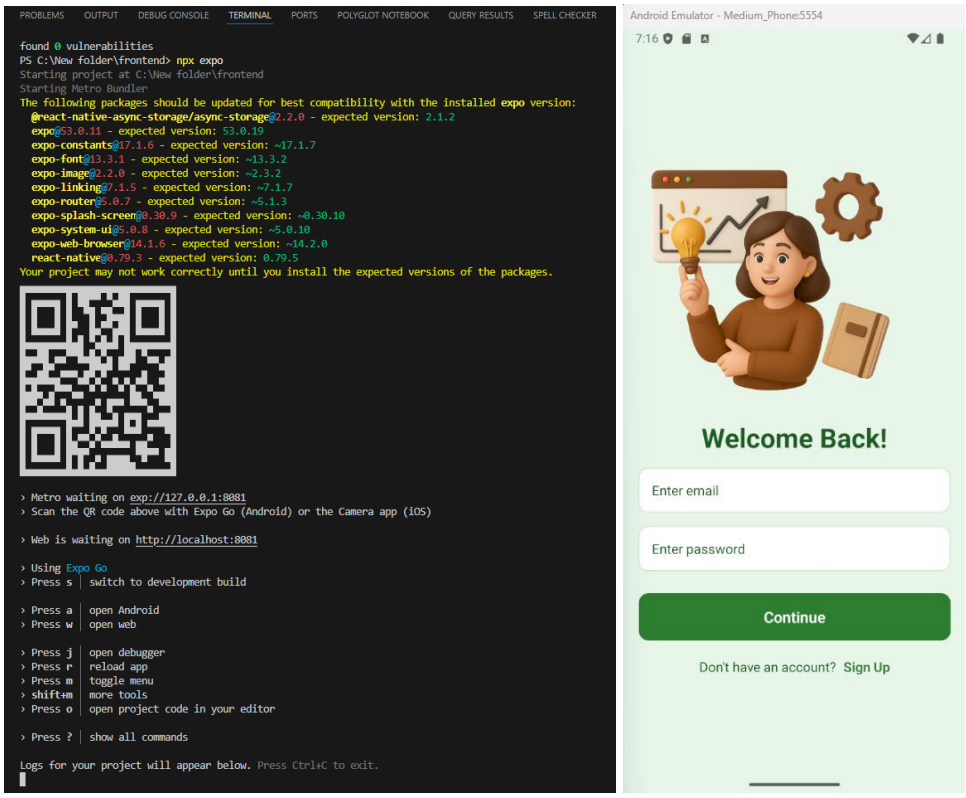


Testing the Front-End React Native project.

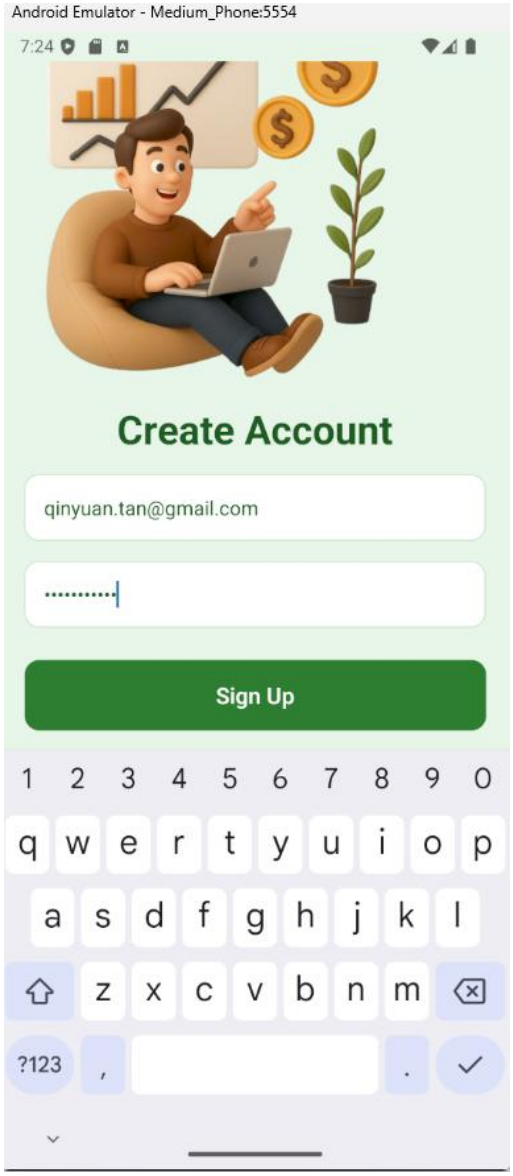
Open your smartphone emulator. Will be using Android Studio in the example.



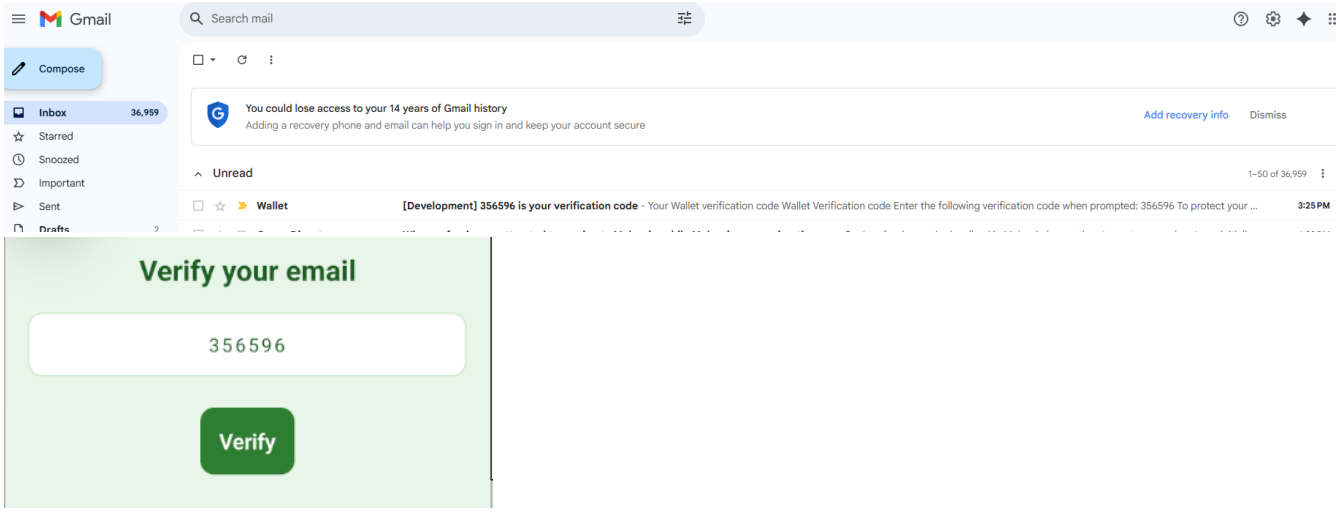
In the Terminal in Visual Studio Code, use the command npx expo to start using Expo. And then start the application in the emulator.



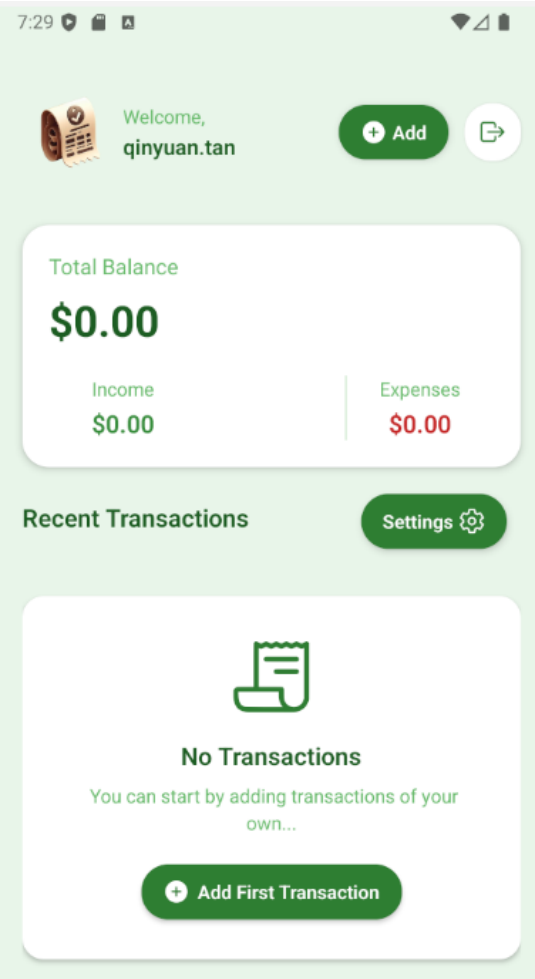
Click sign up, and you will be redirected to the sign up page. Sign up using an exisiting email and create a new password.



Once you had signed up, you will receive a verification email and verify your account.

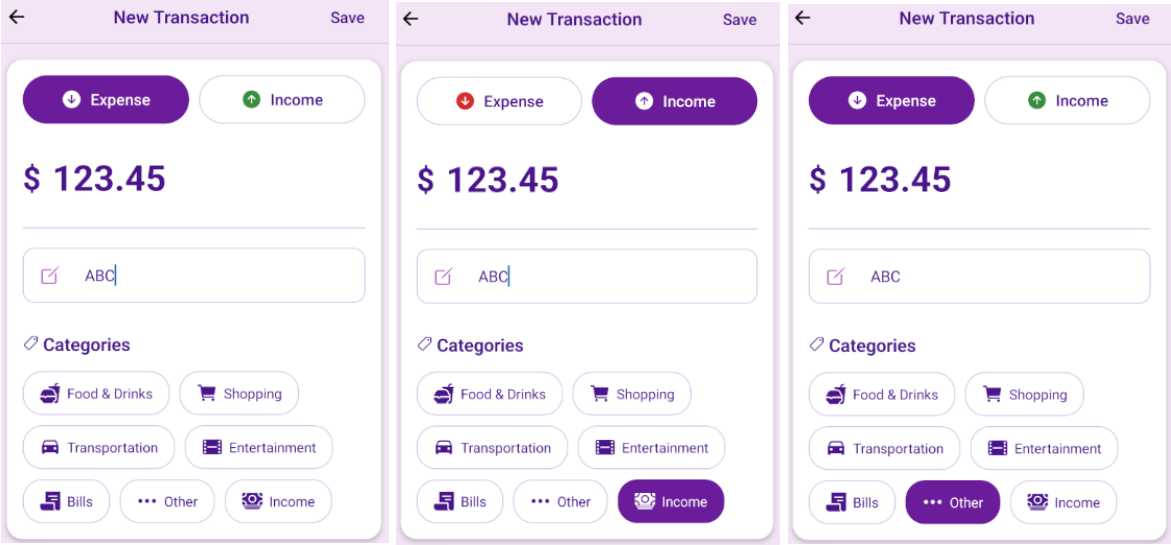


Once verified, you will be directed into the main page. The Transactions component (the component below “Recent Transactions” and the Settings button) may take awhile to load (up to 1 minute) due to the Cloud Application platform requiring time to warm up.

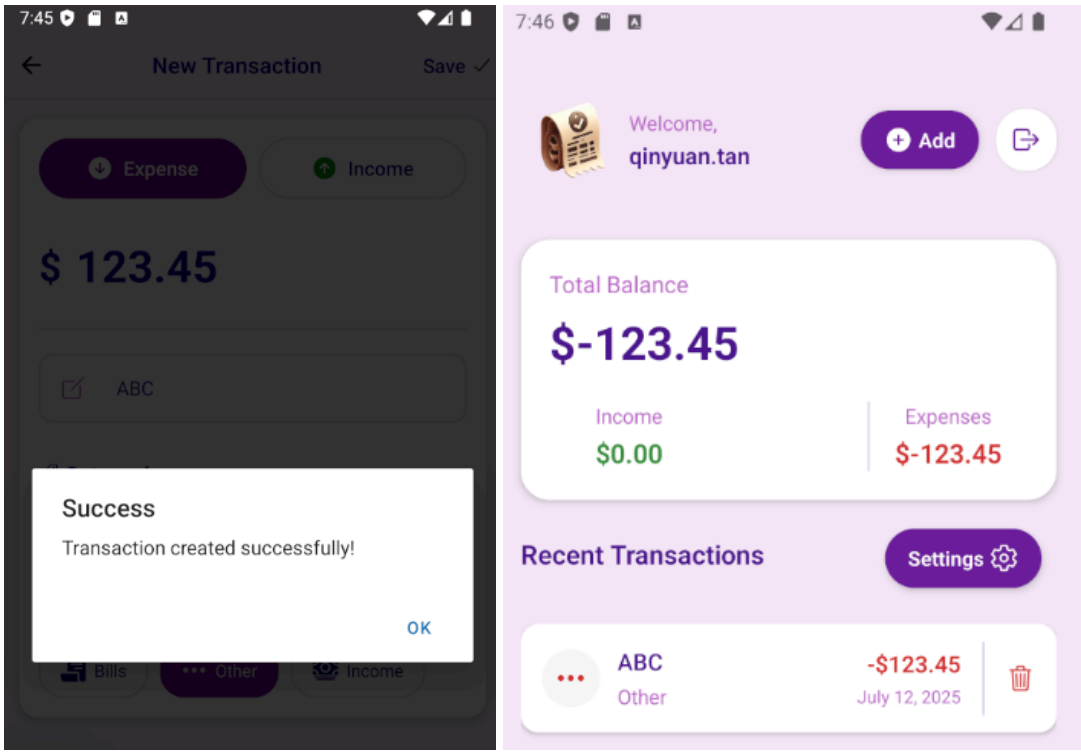


Click settings button to change the colour of the application to your liking.

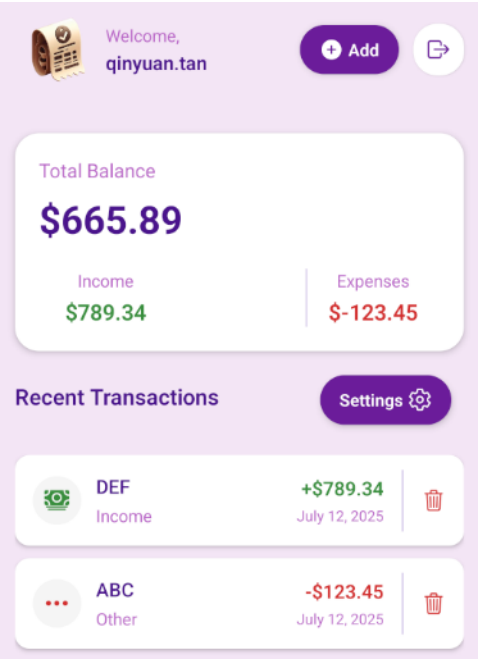
Go back to the main page and click either the Add button or Add First Transaction to create your first recorded transaction. Note that when you select Income at the top, the Income button under Categories will also be automatically selected. And if you click Expense, the Income button under Categories will also be automatically unselected. Also should you select anything else other than Income under Categories, the Income button at the top will automatically unselect and Expense button will be selected instead.



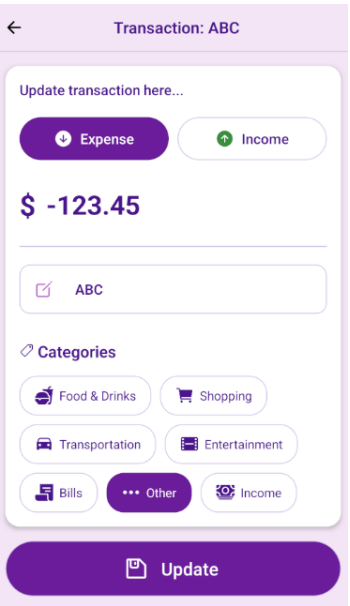
Click Save and then return to the main page. The changes should be reflected under the Balance Component.

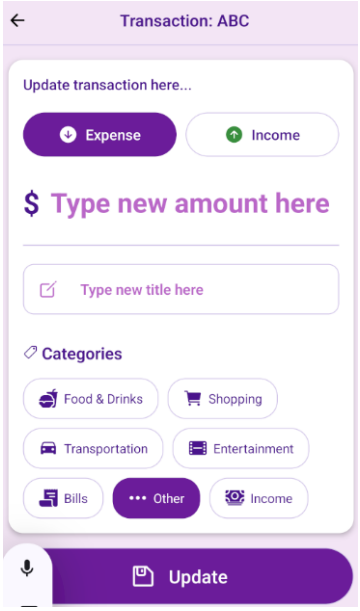


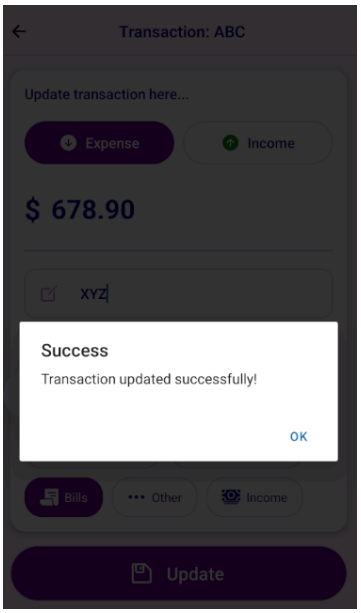
You can also add an Income record to see how it affects the Balance overall.

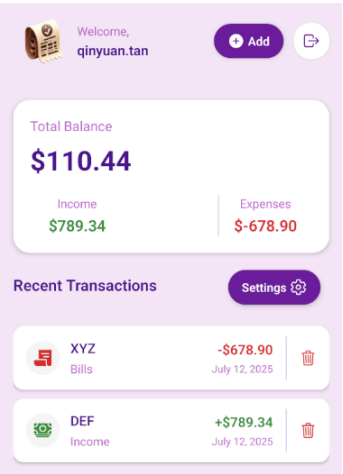


Click on one of the recorded Transactions to update it.









In the IDE, in every 10 minutes you should see this message in the console. It is a background ping that occurs every 10 minutes so as not to be logged out of the database. As long as the user has logged into the application, the background pinging will occur until the user exits the application.

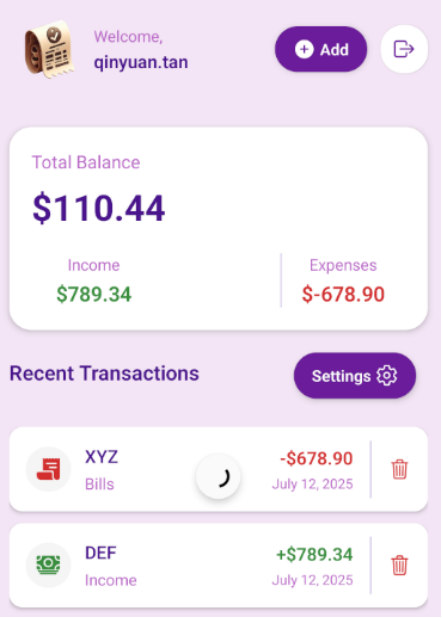
```
Android Bundled 6093ms node_modules\expo-router\entry.js (1266 modules)
WARN Clerk: Clerk has been loaded with development keys. Development instances have strict usage limits and should not be used when deploying your application to produc
tion. Learn more: https://clerk.com/docs/deployments/overview
LOG This will be called every 600 seconds
LOG Cron job's fetch was successful!
```

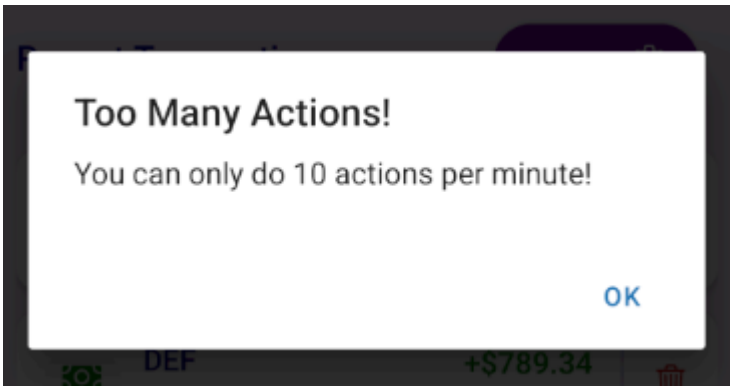
JS api.js | .env | JS background.js 1, M | JS transaction.jsx 8, M

C:\New folder\frontend\constants\api.js

```js
import { useEffect } from 'react'
import {background_API_URL} from "../constants/api";


export function Background()
{
    useEffect(() =>
    {
        const interval = setInterval(() => {

            console.log("This will be called every 600 seconds")

            ping();

        }, 600000);}, []);

}

const ping = async () =>
{
    try
    {
        const response = await fetch(background_API_URL);

        if (response.status === 200)
        {
            console.log("Cron job's fetch was successful!")
        }
    }
    catch (error)
    {
        console.error(error);
    }
}
```

On the main page, try refreshing the transactions more than 10 times in a minute by holding on to any transaction and scrolling down. You should get an alert claiming too many actions.



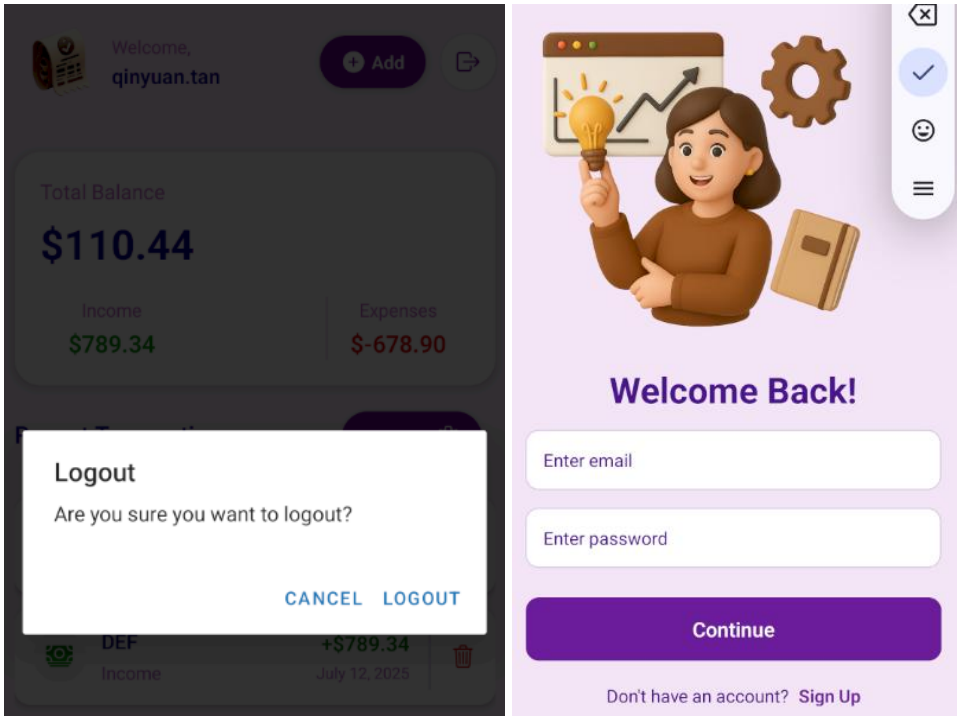


This is due to the Back-End rate limiting function.

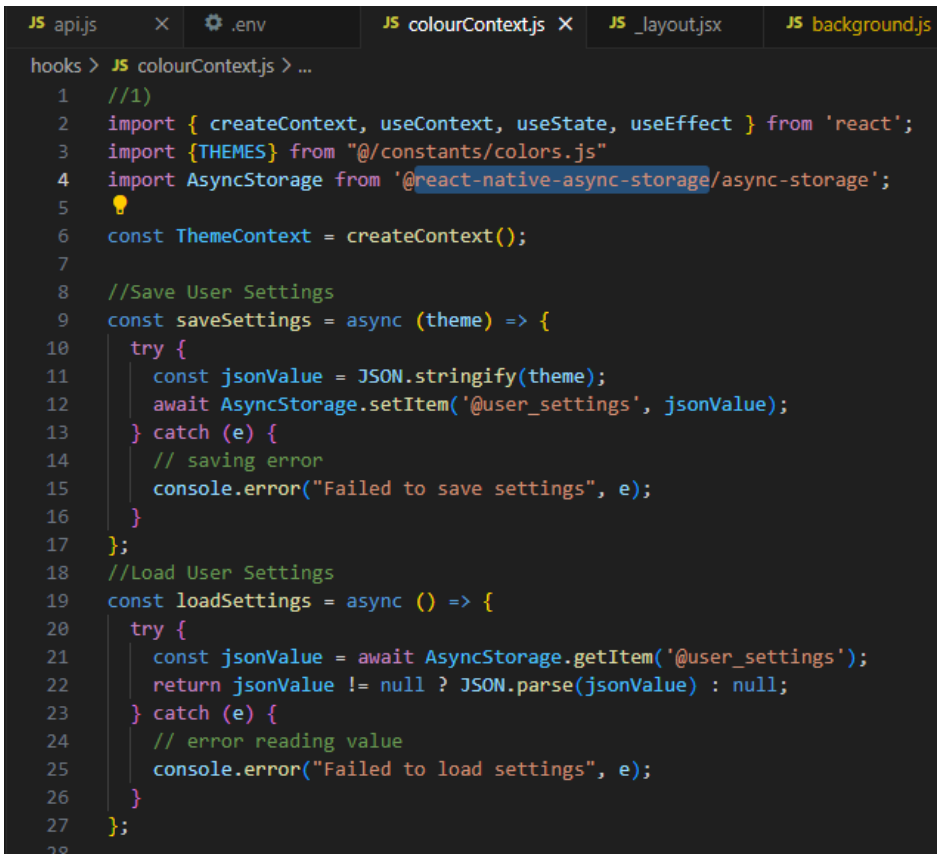
RateLimitingFilter.java × DemoApplication.java TransactionController.java application.properties

```java
package com.example.demo.middleware;

import jakarta.servlet.*;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;

@Component  6 usages  QinYuanMachiavelrous *
public class RateLimitingFilter implements Filter {

    // Map to store request counts per IP address
    private final Map<String, AtomicInteger> requestCountsPerIpAddress = new ConcurrentHashMap<>();  2 usages

    // Maximum requests allowed per minute
    private static final int MAX_REQUESTS_PER_MINUTE = 10;  1 usage

    @Override  QinYuanMachiavelrous
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest httpServletRequest = (HttpServletRequest) request;
        HttpServletResponse httpServletResponse = (HttpServletResponse) response;

        String clientIpAddress = httpServletRequest.getRemoteAddr();

        // Initialize request count for the client IP address
        requestCountsPerIpAddress.putIfAbsent(clientIpAddress, new AtomicInteger( initialValue: 0));
        AtomicInteger requestCount = requestCountsPerIpAddress.get(clientIpAddress);

        // Increment the request count
        int requests = requestCount.incrementAndGet();

        // Check if the request limit has been exceeded
        if (requests > MAX_REQUESTS_PER_MINUTE) {
            httpServletResponse.setStatus(HttpStatus.TOO_MANY_REQUESTS.value());
            httpServletResponse.getWriter().write( s: "Too many requests. Please try again later.");
            return;
        }

        // Allow the request to proceed
        chain.doFilter(request, response);

        // Optional: Reset request counts periodically (not implemented in this simple example)
    }
}
```

In the main page, press the logout button at the top-right and close the application and open the application. Notice that the colour theme of the application is the same as when you exit the application.



This is because of cache created and stored in the smartphone over the colour theme using react-native-async-storage.



```
//1)
import { createContext, useContext, useState, useEffect } from 'react';
import {THEMES} from "@/constants/colors.js"
import AsyncStorage from '@react-native-async-storage/async-storage';

const ThemeContext = createContext();

//Save User Settings
const saveSettings = async (theme) => {
  try {
    const jsonValue = JSON.stringify(theme);
    await AsyncStorage.setItem('@user_settings', jsonValue);
  } catch (e) {
    // saving error
    console.error("Failed to save settings", e);
  }
};
//Load User Settings
const loadSettings = async () => {
  try {
    const jsonValue = await AsyncStorage.getItem('@user_settings');
    return jsonValue != null ? JSON.parse(jsonValue) : null;
  } catch (e) {
    // error reading value
    console.error("Failed to load settings", e);
  }
};
```

```jsx
export const ThemeProvider = ({ children }) => {

    const [theme, setTheme] = useState(THEMES.coffee);

    //Use the Functions for Caches
    const fetchSettings = async () => {
      const userSettings = await loadSettings();
      if (userSettings) {
        setTheme(userSettings);
      }
    };
    const toggleTheme = (newTheme) => {
        setTheme(newTheme);
        saveSettings(newTheme);
    };
    useEffect(() => {
    fetchSettings();
    }, []);
    //Use the Functions for Caches

    return (
        <ThemeContext.Provider value={{ theme, toggleTheme }}>
            {children}
        </ThemeContext.Provider>
    );
};

export const useTheme = () => useContext(ThemeContext);
```

JS api.js | .env | JS colourContext.js | JS _layout.jsx ×

app > JS _layout.jsx > RootLayout

```jsx
import { Slot } from "expo-router";
import { ClerkProvider } from '@clerk/clerk-expo'
import { tokenCache } from '@clerk/clerk-expo/token-cache'
import SafeScreen from "@/components/SafeScreen"
import { StatusBar } from "expo-status-bar";

//2)
import { ThemeProvider } from '../hooks/colourContext';

export default function RootLayout()
{

  //2)
  return (
    <ClerkProvider tokenCache={tokenCache}>
    <ThemeProvider>
      <SafeScreen>
        <Slot />
      </SafeScreen>
      <StatusBar style="dark"/>
    </ThemeProvider>
    </ClerkProvider>
  );
}

```