Velammal College of Engineering and Technology, Madurai

(Autonomous)

Department of Information Technology

21PCS12 – Android App Development Mini Project Report

Name of the Student: Reshmika K S

Year/Sem/Sec: III/VI/B

Roll. No: 21ITB16

Register. No: 913121205077

Project Title: Student Information System – VCET

Student Signature **Faculty Signature**

Project Review

STUDENT INFORMATION SYSTEM

The "Student Information System" app is a pivotal development aimed at revolutionizing how students access and manage their academic information. In today's fast-paced digital era, mobile applications have become indispensable tools for enhancing user experience and convenience, particularly in the education sector. This app, developed using React Native framework with Expo and SQLite, embodies a user-centric approach by providing a seamless and intuitive platform for students to navigate through their academic journey.

At its core, the app offers a secure login mechanism, ensuring that only authorized users can access sensitive information. Once logged in, students are greeted with a well-organized tab navigation system, starting with the Home screen where they can find essential college details, announcements, and contact information. This centralized hub serves as a gateway to the app's various functionalities, promoting easy access to vital information.

The Performance screen stands out as a cornerstone feature, offering students detailed insights into their internal performance metrics. From exam scores to assignment grades and overall performance trends, students can track their academic progress with precision. The seamless integration of SQLite enables real-time data retrieval and storage, ensuring that students have up-to-date access to their performance reports.

In tandem with performance tracking, the app also includes an Attendance screen, allowing students to monitor their attendance records visually. This feature not only promotes accountability but also empowers students to take proactive measures to meet attendance requirements. By providing a clear overview of attendance percentages and trends, the app fosters a culture of responsibility among students.

The app's development journey was guided by a commitment to user-centric design and efficient functionality. Challenges such as data management and interface optimization were met through strategic planning and implementation. User feedback and testing played a crucial role in refining the app's features and ensuring a smooth user experience across different devices and platforms.

Aim

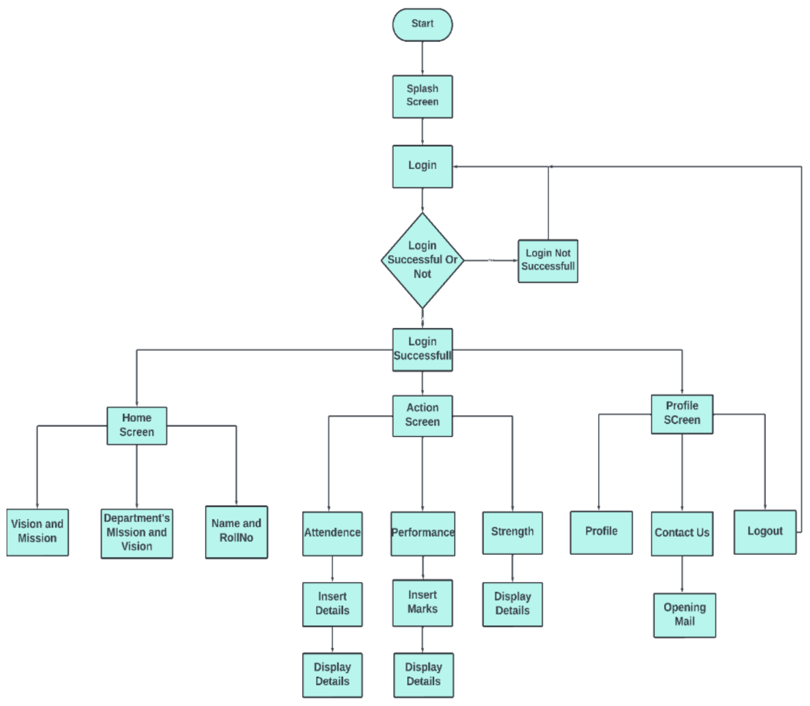
To enhance accessibility and promote accountability by providing students with easy access to their academic information. The app aims to streamline data management, improve user experience, and foster a connected learning environment for improved academic outcomes and student satisfaction.

Introduction

The "Student Information System" app is a comprehensive solution developed using React Native with Expo and SQLite for the backend, designed to improve access to internal performance and attendance reports for students. The app's login screen ensures secure user authentication, while the home screen provides essential college details. The performance screen offers options to check internal metrics and attendance, including student strength data. Users can manage their profiles and contact the college via email through the app.

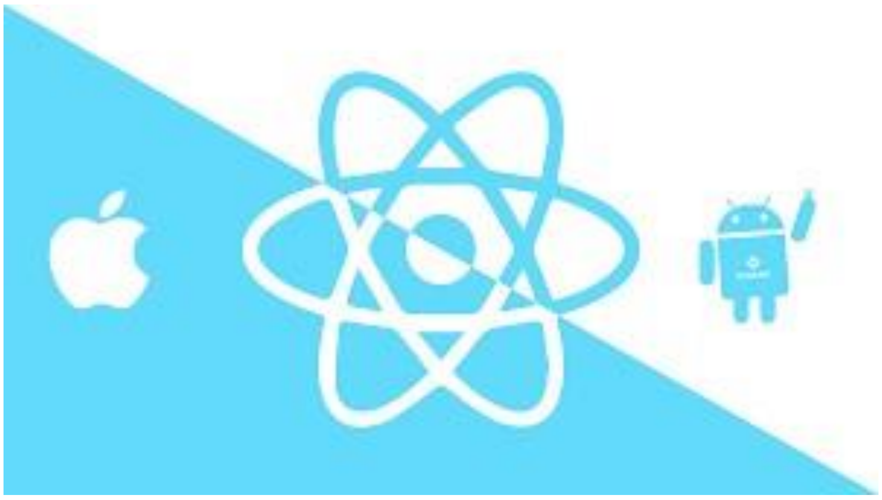
React Native was chosen for its cross-platform capabilities and robust performance, while Expo simplified testing and deployment processes. SQLite handles backend data storage and management efficiently. The development process involved careful design considerations, user interface development, seamless integration with SQLite for data management, and rigorous testing procedures to ensure functionality and usability. Challenges were addressed through strategic planning and iterative improvements. User feedback and testing results have shown positive outcomes in terms of usability, performance, and user satisfaction. In conclusion, the app has met its objectives of enhancing accessibility, promoting accountability, and fostering a connected learning environment. Future enhancements could include additional features for academic resource access and collaboration tools. References include resources, libraries, and frameworks used in the app's development.

Flow Diagram



Concepts Involved

React Native Framework: React Native stands out as a leading framework for mobile app development due to its capability to create native-like experiences across multiple platforms, including iOS and Android. Developers leverage React, a JavaScript library, to construct intuitive user interfaces (UIs). This approach not only streamlines development but also ensures a consistent look and feel for users regardless of the platform they're on. By writing code once, developers can efficiently deploy it across various platforms, saving time and resources in the development lifecycle.



Component-Based Architecture: React Native's architecture revolves around reusable components that encapsulate UI elements. Components represent discrete parts of the interface, such as buttons, text inputs, or entire screens. This modular design fosters code reusability, simplifies maintenance, and enhances scalability. Developers can compose complex UIs by combining and nesting components, resulting in a highly flexible and efficient development process.

Navigation: Navigation plays a pivotal role in guiding users through an app's different screens and sections seamlessly. React Navigation, a popular navigation library for React Native, offers various navigation patterns like stack, tab, and drawer navigation. These navigation paradigms ensure intuitive user experiences, allowing users to navigate effortlessly between screens. Through well-structured navigation, developers can enhance user engagement and retention within the app.

User Authentication: Security is paramount in mobile applications, particularly concerning user authentication. React Native apps implement robust authentication mechanisms to safeguard user data and application features. Concepts such as authentication tokens, secure storage, and encryption are pivotal in verifying user credentials securely. By adhering to best practices in authentication flow, developers ensure that users can access the app's features securely and seamlessly.

Data Persistence: Data persistence ensures that user preferences and application data remain accessible even when offline. React Native offers storage solutions like AsyncStorage for simple key-value pair storage and SQLite for more complex relational data storage. By employing techniques such as data serialization, encryption, and offline caching, developers guarantee data integrity and availability, providing users with a seamless experience irrespective of network connectivity.

UI/UX Design: UI/UX design plays a pivotal role in shaping the overall user experience and interface aesthetics. React Native developers apply principles of responsive layout design, navigation patterns, typography, color schemes, and iconography to create visually appealing and intuitive interfaces. Leveraging design libraries like React Native Elements or NativeBase, developers ensure consistency and aesthetic appeal across the application, enhancing user engagement and satisfaction.

About SQLite

SQLite is a relational database management system (RDBMS) that implements a small, fast, self-contained, high-reliability, full-featured SQL database engine. Here are some key features of SQLite:

- **Serverless:** Unlike traditional client-server databases, SQLite does not require a separate server process. It operates as a library that directly accesses the database file.
- **Zero Configuration:** SQLite does not have a separate server process that needs to be set up, configured, or managed. It operates seamlessly with minimal configuration.
- **ACID Transactions:** SQLite supports ACID (Atomicity, Consistency, Isolation, Durability) transactions, ensuring data integrity and reliability.
- **Cross-Platform:** SQLite databases are compatible with various operating systems, including iOS, Android, Windows, and macOS, making them highly versatile.

Why SQLite?

SQLite is a lightweight, serverless, self-contained, and embedded SQL database engine. It's a popular choice for mobile app development due to its simplicity, efficiency, and compatibility with various platforms. Here are some reasons why SQLite was chosen for the backend of this app:

- **Portability:** SQLite databases are stored in a single file, making them highly portable across different platforms and devices.
- **Ease of Use:** SQLite is easy to set up and integrate into mobile apps. Its SQL-based interface is familiar to developers, allowing for seamless database operations.
- **Performance:** SQLite is optimized for performance, providing fast read and write operations even on resource-constrained devices.
- **Compatibility:** SQLite is supported by Expo, a popular framework for building cross-platform mobile apps, making it an ideal choice for React Native apps.

SQLite with React Native Expo

In this app, SQLite is used in conjunction with React Native Expo for building cross-platform mobile applications. Here's how SQLite is integrated into the app:

1. **Database Initialization:** The **expo-sqlite** package is used to open a connection to the SQLite database (**vcet.db**) during app initialization.
2. **Table Creation:** Upon initialization, the app checks if the required tables exist in the database. If not, it creates the necessary tables, such as the 'cse' table, to store user data.
3. **Data Manipulation:** The app performs various database operations, such as inserting sample records, retrieving user details, and executing SQL queries to interact with the database.
4. **Asynchronous Transactions:** Database transactions are executed asynchronously to ensure smooth performance and responsiveness of the app.

Integrating SQLite with React Native Expo provides a convenient solution for mobile app development. Expo's ecosystem offers tools and libraries that simplify SQLite integration, streamlining the process of incorporating a local database into React Native projects. This combination ensures compatibility, efficiency, and optimal performance for mobile applications.

Advantages of Using SQLite

SQLite offers several advantages that make it a preferred choice for mobile app development:

- **Portability:** SQLite databases are self-contained and stored in a single file, making them highly portable across different platforms and devices. This simplifies deployment and distribution of mobile apps.
- **Ease of Use**: SQLite is easy to set up and integrate into mobile apps. Its SQL-based interface is familiar to developers, allowing for seamless database operations without the need for complex setup or configuration.
- **Zero Configuration**: Unlike client-server databases, SQLite does not require a separate server process. It operates as a library that directly accesses the database file, eliminating the need for server setup and configuration.
- **Efficiency:** SQLite is optimized for efficiency and performs well even on resource-constrained devices. It provides fast read and write operations, ensuring smooth performance and responsiveness of mobile apps.

The backend setup using SQLite in this app offers a reliable and efficient solution for storing and managing data. By leveraging the advantages of SQLite, the app achieves seamless database integration, optimal performance, and a consistent user experience across different platforms.SQLite's portability, ease of use, efficiency, and cross-platform compatibility make it well-suited for mobile app development. With its support for ACID transactions and low overhead, SQLite ensures data integrity, reliability, and cost-effectiveness in handling critical data and transactions.

Setup:

To set up the backend for your React Native Expo app using SQLite, follow these steps:

1. **Install Dependencies:**

   Ensure you have Expo CLI installed globally on your machine. If not, you can install it using npm or yarn:

   ```
   npm install -g expo-cli
   # or
   yarn global add expo-cli
   ```

2. **Create a New Expo Project:**
   Create a new Expo project by running the following command and following the prompts:

   ```
   expo init vcet
   ```

3. **Install SQLite Package:**
   Install the SQLite package for Expo using npm or yarn:

   ```
   expo install expo-sqlite
   ```

4. **Set Up the SQLite Database:**
   Create your SQLite database file within your project. You can use the openDatabase function from expo-sqlite to connect to the database. Example:

   ```
   import { openDatabase } from 'expo-sqlite';
   const db = openDatabase('vcet.db');
   ```

5. **Define Database Schema:**
   - Define your database schema, including tables and their columns.
   - You can execute SQL statements to create tables within your SQLite database.

   ```
   // Create tables if not exists and insert sample records
   export const initDatabase = () => {
     db.transaction(tx => {
       tx.executeSql(
         `CREATE TABLE IF NOT EXISTS cse (
           department TEXT,
           batch TEXT,
           section TEXT,
           rollNumber TEXT PRIMARY KEY,
           name TEXT
         )`,
         [],
         () => {
           console.log('Table created successfully');
   ```

```
        insertSampleRecords(); // Call function to insert sample records after table
creation
      },
      (_, error) => console.error('Error creating table:', error)
    );
  });
};
```

Functions With Back End:

1.initDatabase Function:

- **Purpose:** This function initializes the SQLite database by creating the necessary table if it doesn't exist and inserts sample records into it.
- **Database Initialization:** Checks if the 'cse' table exists, and if not, creates it with the required columns.
- **Table Creation Success:** Logs a message indicating successful table creation.
- **Sample Records Insertion:** Calls the **insertSampleRecords** function after creating the table to insert sample records.
- **Error Handling:** Logs an error message if there's an issue creating the table.
- **Transactions:** Utilizes transactions to ensure atomicity and consistency while executing SQL statements.
- **Asynchronous Operation:** Executes database operations asynchronously to prevent blocking the main thread.

2.insertSampleRecords Function:

- **Purpose:** Inserts sample records into the 'cse' table to populate it with initial data.
- **Sample Record Definition:** Defines an array of sample records, each containing department, batch, section, roll number, and name.
- **Looping Through Records:** Iterates over each sample record and inserts it into the 'cse' table.
- **Ensuring Data Integrity:** Utilizes transactions to ensure all records are inserted atomically to maintain data integrity.
- **Error Handling:** Handles any errors that may occur during the insertion process.
- **Data Consistency:** Ensures consistent formatting and data types for each record field to prevent data inconsistencies.
- **Testing:** Validates the insertion process by checking the database for the presence of inserted records.

```
// Function to insert sample records into the database
const insertSampleRecords = () => {
  const sampleRecords = [
    { department: 'CSE', batch: '2015-2019', section: 'A', rollNumber: '15CSE01',
name: 'Abi Ranjeetha A R' },
  { department: 'CSE', batch: '2015-2019', section: 'A', rollNumber: '15CSE02', name: 'Abinaya S' },
  { department: 'CSE', batch: '2015-2019', section: 'A', rollNumber: '15CSE03', name: 'Anu B' },
  { department: 'CSE', batch: '2015-2019', section: 'A', rollNumber: '15CSE04', name: 'Deepika E' },
……
 ];

 insertRecords(sampleRecords);
};
```

3.insertRecords Function:

- **Purpose:** Inserts records into the 'cse' table based on the provided data.
- **Parameter**: Accepts an array of records to be inserted into the database.
- **Looping Through Records**: Iterates over each record in the array and inserts it into the 'cse' table.
- **Transaction Management:** Executes the insertion operation within a transaction to ensure atomicity and data consistency.
- **Error Handling:** Handles any errors that may occur during the insertion process.
- **Data Validation:** Validates the data integrity before insertion to prevent database corruption.
- **Scalability:** Designed to handle large datasets efficiently by batching insertions within transactions.

```
// Function to insert records into the database
export const insertRecords = (records) => {
 db.transaction(tx => {
  records.forEach(record => {
   tx.executeSql(
      'INSERT INTO cse (department, batch, section, rollNumber, name) VALUES (?, ?, ?, ?, ?)',
       [record.department, record.batch, record.section, record.rollNumber, record.name]
   );
  });
 }); };
```

4.getAllCSERecords Function:

- **Purpose:** Retrieves all records from the 'cse' table.
- **Query Execution:** Executes an SQL query to select all records from the 'cse' table.
- **Promise-Based Operation:** Returns a Promise to asynchronously handle the result of the query.

- **Row Extraction:** Extracts the result rows from the query response for further processing.
- **Data Resolution:** Resolves the Promise with the array of retrieved records for consumption by the caller.
- **Error Handling:** Rejects the Promise if there's an error executing the query or processing the result.
- **Asynchronous Nature:** Executes database operations asynchronously to avoid blocking the main thread and improve performance.

```
// Query to retrieve all records from cse table
export const getAllCSERecords = () => {
  return new Promise((resolve, reject) => {
    db.transaction(tx => {
      tx.executeSql(
        'SELECT * FROM cse',
        [],
        (_, { rows }) => {
          resolve(rows._array);
        },
        (_, error) => {
          reject(error);
        }
      );
    });
  });
};
```

5.fetchAndDisplayRecords Function:

- **Purpose:** Fetches and displays records from the 'cse' table for testing or debugging purposes.
- **Asynchronous Operation:** Utilizes async/await to handle asynchronous database operations.
- **Error Handling**: Catches and logs any errors that occur during the fetching process.
- **Testing Convenience**: Provides a convenient way to test database retrieval functionality.
- **Logging:** Logs the retrieved records to the console for inspection.
- **Integration Testing:** Can be integrated into unit tests to validate database functionality.
- **Debugging Aid:** Helps identify issues with database queries or data retrieval logic during development.

```
// Fetch and display records from the database
```

```
export const fetchAndDisplayRecords = async () => {
  try {
    const records = await getAllCSERecords();
     console.log(records); // Display the retrieved records in the console or use
them as needed
  } catch (error) {
    console.error('Error fetching records:', error);
  }
};
```

6. **handleLogin Function :**
   - **Purpose:** This function handles the login process by querying the SQLite database to verify the user's credentials.
   - **Transaction Execution:** Uses a transaction to execute the SQL query in a safe and consistent manner.
   - **Query Execution:** Executes an SQL SELECT query to retrieve records from the 'cse' table based on the provided username and password.
   - **Parameterized Query:** Uses parameterized queries to prevent SQL injection attacks and securely pass user inputs (username and password) to the database.
   - **Result Handling:** Processes the result of the query callback function, which contains the retrieved rows from the database.
   - **User Authentication:** Checks if any rows are returned from the query result. If there are rows, it means that the username and password match an existing record in the database, and the user is authenticated.
   - **Navigation:** If authentication is successful, navigates to the 'TabNavigation' screen, passing the username as a parameter to display personalized content.
   - **Error Handling:** Logs any errors that occur during the query execution process, such as database connection issues or syntax errors in the SQL query.
   - **Alert for Invalid Credentials:** If no rows are returned from the query result, it means that the username or password is incorrect. In this case, displays an alert to inform the user of invalid credentials and prompt them to try again.
   - **Secure Navigation:** Uses navigation to switch to the 'TabNavigation' screen only if the user credentials are verified. Otherwise, stays on the login screen to allow the user to retry authentication.

```
const handleLogin = () => {
  db.transaction(tx => {
    tx.executeSql(
      'SELECT * FROM cse WHERE name = ? AND rollNumber = ?',
      [username, password],
      (_, { rows }) => {
        if (rows.length > 0) {
          const user = rows.item(0);
          navigation.navigate('TabNavigation', { username: user.name });
        } else {
          alert('Invalid username or password. Please try again.');
        }
      },
      (_, error) => {
        console.error('Error fetching user:', error);
      }
    );
  });
};
```

Module Description

Login Screen:

Description:

The Login component is a crucial part of the React Native application, responsible for handling user authentication.

It offers a visually appealing and user-friendly interface for users to log in to the app securely.

Functionality:

- Displays the app's logo and a heading ("V C E T") for brand identification.
- Provides input fields for users to enter their name and password.
- Implements navigation using React Navigation's useNavigation hook to redirect users to the main content screen (TabNavigation) upon successful login.
- Performs basic input validation and displays error messages if the user enters incorrect credentials.
- Incorporates a login button styled with a background color, rounded corners, and a contrasting text color for a prominent call-to-action.

Dependencies Used:

- React Native: The core framework for building mobile applications.
- @react-navigation/native: A navigation library used for managing navigation within the app.

Components:

View: Serves as the main container for structuring the layout of the login screen.
Text: Displays various text elements, including the app heading, error messages, and button text.
TextInput: Allows users to input their name and password securely.
Image: Renders the app's logo at the center of the screen, enhancing brand visibility.

TouchableOpacity: Wraps the login button, providing touch interaction for better user experience.
StyleSheet: Defines styles for components using CSS-like syntax, ensuring a consistent and visually appealing design.

Home Screen:

Description:

The HomeScreen component is a pivotal aspect of the React Native application, primarily focusing on presenting essential information and features to users accessing the app's home interface.

Functionality:

- Showcases the app's logo and the college's name ("Velammal College of Engineering and Technology") for brand recognition and identification.
- Displays images representing various departments of the college along with their names.
- Includes sections highlighting the college's accolades, such as NBA, NAAC, ARIIA, and NIRF recognitions.
- Provides a visually appealing layout with responsive design elements for a seamless user experience.

Dependencies Used:

- React Native: Core framework for developing mobile applications.
- @react-navigation/native: Navigation library for managing app navigation.
- colors (utils/colors): Utilized for maintaining consistent color schemes across the app.

Components:

View: Essential for structuring the layout of the home screen, including header, sections, and department cards.
Text: Displays text elements such as college name, section titles, and department names.
Image: Renders images representing departments, college accolades, and the app's logo.
ScrollView: Enables scrolling functionality for content that exceeds the screen size.
TouchableWithoutFeedback: Wraps department cards to provide touch interaction.
StyleSheet: Defines styles for components using CSS-like syntax, ensuring a cohesive and visually appealing design.

Profile Screen:

Description:

The ProfileScreen component is a vital part of the React Native application, focusing on displaying user profile information and providing essential functionalities such as logout and contacting support.

Functionality:

- Displays the app's logo and a personalized welcome message ("Welcome USER") for user identification and engagement.
- Presents user details such as name, department, section, and roll number in an organized layout (innercontainer).
- Includes buttons for logout and contacting support (handleContactUs) via email (Linking.openURL('mailto:principal@vcet.ac.in')).
- Utilizes icons from the Expo vector icons library (AntDesign and Entypo) for visual enhancements.

Dependencies Used:

- React Native: Core framework for developing mobile applications.
- @expo/vector-icons: Provides access to a library of customizable icons for UI design.
- **Linking (built-in): Enables functionality to open external links such as email clients.**

Components:

View: Essential for structuring the layout of the profile screen, including sections for user details, buttons, and logo display.
Text: Displays text elements such as welcome message, user details, and button labels.
Image: Renders the app's logo for brand recognition and visual appeal.
TouchableOpacity: Wraps buttons to provide touch interaction for logout and contacting support.
StyleSheet: Defines styles for components using CSS-like syntax, ensuring a cohesive and visually appealing design.

Action Screen:

Description:

The ActionScreen component is a pivotal part of the React Native application, focusing on providing users with quick access to essential actions such as viewing attendance reports, performance reports, and student strength information.

Functionality:

- Utilizes React Navigation's useNavigation hook to handle navigation between different screens within the app.
- Includes buttons for accessing attendance reports (handleAttendancePress), performance reports (handlePerformancePress), and student strength information (handleStudentsPress).
- Each button is associated with an icon and text label for intuitive user interaction and navigation.

Dependencies Used:

- React Native: Core framework for developing mobile applications.
- @react-navigation/native: Navigation library used for managing navigation within the app.
- @expo/vector-icons: Provides access to a library of customizable icons for UI design.

Components:

View: Essential for structuring the layout of the action screen, including sections for each action button.
Text: Displays text elements such as action button labels for clear user understanding.
TouchableOpacity: Wraps each action button to provide touch interaction and trigger navigation events.
StyleSheet: Defines styles for components using CSS-like syntax, ensuring a cohesive and visually appealing design.

TabNavigation

Description:

The TabNavigation component is a crucial aspect of the React Native application's navigation system, utilizing React Navigation's Bottom Tab Navigator (createBottomTabNavigator) to facilitate navigation between different screens within the app.

Functionality:

- Initializes a Bottom Tab Navigator (Tab.Navigator) to manage navigation between three main screens: HomeScreen, ActionScreen, and ProfileScreen.
- Configures screen options such as active tab color (tabBarActiveTintColor) using custom colors from the colors utility file (colors.PRIMARY).
- Defines individual tab screen components (Tab.Screen) with associated labels and icons for each tab.
- Utilizes icons from the Expo vector icons library (Entypo, AntDesign, FontAwesome5) to enhance visual representation for tabs.

Dependencies Used:

- @react-navigation/bottom-tabs: Bottom Tab Navigator used for creating a tab-based navigation interface.
- @expo/vector-icons: Provides access to a library of customizable icons for UI design.
- colors (utils/colors): Utilized for maintaining consistent color schemes across the app.

Components:

View: Essential for structuring the layout of the tab navigator and its associated screens.

Text: Displays tab labels using custom styles for color, font size, and margin to ensure readability and aesthetics.
createBottomTabNavigator: Creates a bottom tab navigator to manage navigation between screens.

Action Screen Navigation:

Description:

The ActionScreenNavigation component is a critical part of the React Native application's navigation system. It utilizes React Navigation's Stack Navigator to manage navigation between different screens related to actions and functionalities within the app.

Functionality:

Initializes a Stack Navigator (Stack.Navigator) within a Navigation Container (NavigationContainer) to handle navigation between screens.

- Specifies the initial route name as "Action," indicating that the ActionScreen component will be the initial screen displayed to the user.
- Includes screen components such as ActionScreen and Attendance, allowing users to navigate between these screens seamlessly.
- Provides a structured navigation flow for users to access specific functionalities such as viewing attendance reports (Attendance) from the main action screen (ActionScreen).

Dependencies Used:

- @react-navigation/native: Navigation container used for managing navigation within the app.
- @react-navigation/stack: Stack Navigator used for managing a stack-based navigation flow between screens.

Components:

NavigationContainer: Wraps the entire navigation structure, providing a context for navigation actions and state management.
Stack.Navigator: Configures the stack-based navigation flow, defining screen components and initial routes.
Stack.Screen: Registers individual screens within the stack navigator, associating screen names with their corresponding components (ActionScreen, Attendance, etc.).

Explanation

1. **Imports**: This section imports necessary modules from React Native and other custom components of the application.

2. **Stack Navigator Configuration**: A stack navigator from **@react-navigation/stack** is used to manage navigation between screens in a stack-based manner. The **createStackNavigator** function creates a stack navigator instance called **Stack**.
3. **App Component**: The **App** component is defined as a functional component. It serves as the root component of the application.
4. **Navigation Container**: The **NavigationContainer** component from **@reactnavigation/native** is used to wrap the entire application. It provides the navigation context for the navigation hierarchy.
5. **Stack Navigator Setup**: Within the **NavigationContainer**, a **Stack.Navigator** component is defined to manage the stack navigation. The **initialRouteName** prop is set to **"SplashScreen"**, indicating that the initial route to be displayed is the **SplashScreen**.
6. **Screen Components**: Each screen component is defined within a **Stack.Screen** component. Each screen is associated with a unique name (**name** prop) and the corresponding component (**component** prop). Additionally, the **options** prop is used to configure navigation options for each screen. In this case, **headerShown: false** is used to hide the header for each screen.
7. **Export**: The **App** component is exported as the default export, making it accessible for rendering in the main entry point of the application.

Back – End :

7. HomeScreen :

- **State Initialization:** Initializes the userDetails state variable using the useState hook. This state will store the details of the user retrieved from the database.
- **Effect Hook:** Utilizes the useEffect hook to perform side effects when the component mounts or when the route.params change.
- **Extracting Username:** Destructures the username from the route.params object.
- **Database Transaction:** Executes an SQL SELECT query within a database transaction. The query selects all columns from the 'cse' table where the 'name' column matches the provided username.
- **Query Result Handling:** Processes the result of the query callback function. If the query returns rows (i.e., rows.length > 0), it means that a user with the specified username exists in the database. In this case, it sets the userDetails state with the first row item from the query result.
- **Error Handling:** Logs any errors that occur during the database transaction, such as connection issues or syntax errors in the SQL query.
- **Dependency Array:** Includes route.params in the dependency array of the useEffect hook to re-run the effect whenever the username parameter in the route

changes. This ensures that the user details are fetched from the database whenever the username parameter is updated.

```
const HomeScreen = ({ route }) => {
 const [userDetails, setUserDetails] = useState(null);
 useEffect(() => {
   const { username } = route.params;
   db.transaction(tx => {
     tx.executeSql(
       'SELECT * FROM cse WHERE name = ?',
       [username],
       (_, { rows }) => {
         if (rows.length > 0) {
           setUserDetails(rows.item(0));
         }
       },
       (_, error) => {
         console.error('Error fetching user details:', error);
       }
     );
   });
 }, [route.params]);
```

8. Profile Screen

This ProfileScreen component retrieves and displays user details from the SQLite database based on the provided username parameter.

Here's a breakdown of the code:

- **Route Parameter Extraction:** Destructures the username parameter from the route.params object, which is passed to the component through navigation.
- **State Initialization:** Initializes the userDetails state variable using the useState hook. This state will store the details of the user retrieved from the database.
- **Navigation Object:** Retrieves the navigation object using the useNavigation hook from React Navigation. This object allows for programmatic navigation within the app.
- **Effect Hook:** Utilizes the useEffect hook to perform side effects when the component mounts or when the username parameter changes.
- **Fetch User Details**: Defines a function fetchUserDetails to retrieve user details from the database. Inside this function, a transaction is executed to select all columns from the 'cse' table where the 'name' column matches the provided username. If a user is found (rows.length > 0), the details of the first user are extracted and stored in the userDetails state.
- **Error Handling**: Logs any errors that occur during the database transaction.

- **Logout Handling**: Defines a function handleLogout to navigate back to the login screen when the user initiates a logout action. It uses the navigation.navigate method to navigate to the 'Login' screen.
- **Dependency Array:** Includes username in the dependency array of the useEffect hook to re-run the effect whenever the username parameter changes. This ensures that user details are fetched from the database whenever the username parameter is updated.

```
const ProfileScreen = ({ route }) => {
  const { username } = route.params; // Extract the username from the route params
  const [userDetails, setUserDetails] = useState(null); // State to store user details
  const navigation = useNavigation();

  useEffect(() => {
    // Fetch user details from the database when the component mounts
    const fetchUserDetails = () => {
      db.transaction(tx => {
        tx.executeSql(
          'SELECT * FROM cse WHERE name = ?',
          [username],
          (_, { rows }) => {
            if (rows.length > 0) {
              const user = rows.item(0);
              setUserDetails(user); // Update state with fetched user details
            }
          },
          (_, error) => {
            console.error('Error fetching user details:', error);
          }
        );
      });
    };

    fetchUserDetails(); // Call the fetchUserDetails function
  }, [username]);
  const handleLogout = async () => {
    try {
      // Navigate back to the login screen
      navigation.navigate('Login');
    } catch (error) {
      console.error('Error logging out:', error);
    }
  };
```

9. **About Data :**

- This is sample data in JSON format representing records from the 'cse' table.
- Each record includes details such as 'batch', 'department', 'name', 'rollNumber', and 'section'.

[{"batch": "2015-2019", "department": "CSE", "name": "Abi Ranjeetha A R", "rollNumber": "15CSE01", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Abinaya S", "rollNumber": "15CSE02", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Anu B", "rollNumber": "15CSE03", "section": "A"}, {"batch": "Kowsalya V", "rollNumber": "15CSE16", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Minu P", "rollNumber": "15CSE17", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Mohana Priya T", "rollNumber": "15CSE18", "section": "A"}, {"batch": "2015-2019", "de"Kowsalya V", "rollNumber": "15CSE16", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Minu P", "rollNumber": "15CSE17", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name":iya T", "rollNumber": "15CSE18", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Nivetha Sree S", "rollNumber": "15CSE19", "section": "A"}]

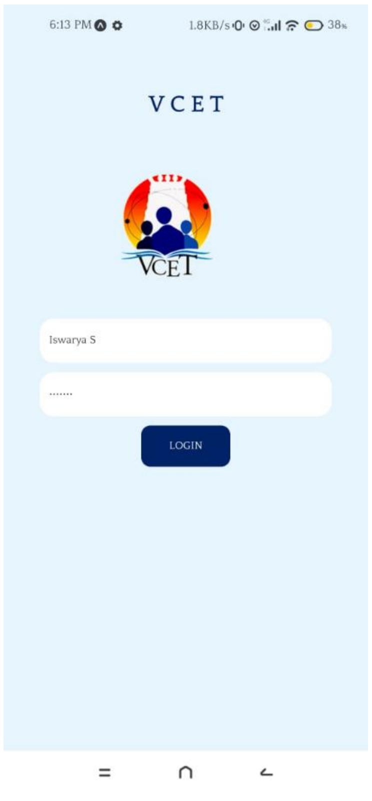
Here's a brief explanation of the data:

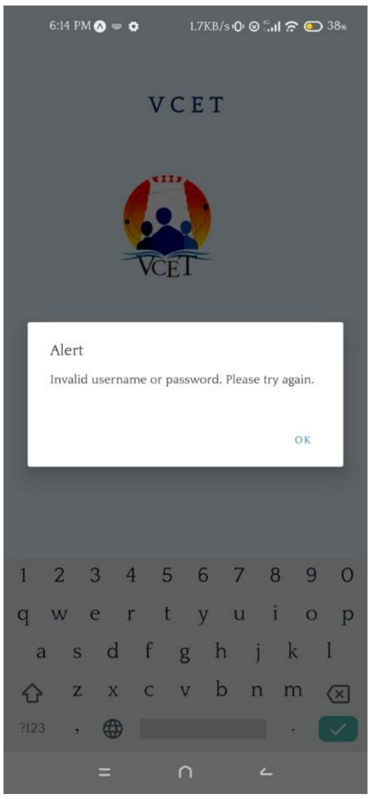
1. **Batch:** Represents the batch year of the student.
2. **Department:** Indicates the department of the student, in this case, 'CSE' (Computer Science and Engineering).
3. **Name:** The full name of the student.
4. **Roll Number:** Unique identifier for each student.
5. **Section:** Denotes the section/class in which the student belongs.

- The provided data appears to be a list of objects, where each object represents a single student's record with the aforementioned attributes.
- This data can be used to populate the 'cse' table in the SQLite database, as demonstrated in code snippets where you insert sample records into the database

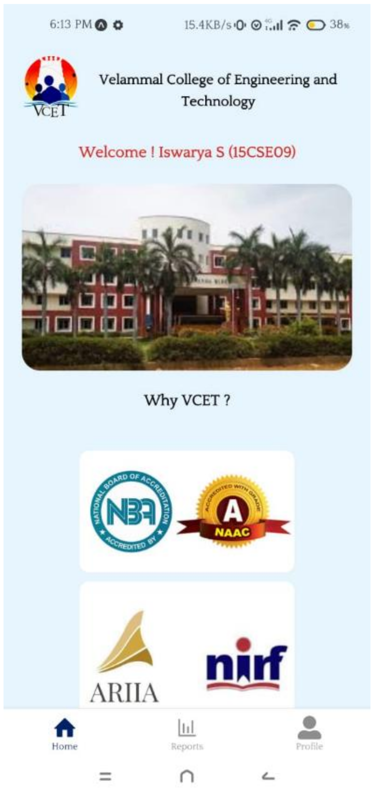
The backend setup using SQLite in this app offers a reliable and efficient solution for storing and managing data. By leveraging the advantages of SQLite, the app achieves seamless database integration, optimal performance, and a consistent user experience across different platforms.SQLite's portability, ease of use, efficiency, and cross-platform compatibility make it well-suited for mobile app development. With its support for ACID transactions and low overhead, SQLite ensures data integrity, reliability, and cost-effectiveness in handling critical data and transactions.

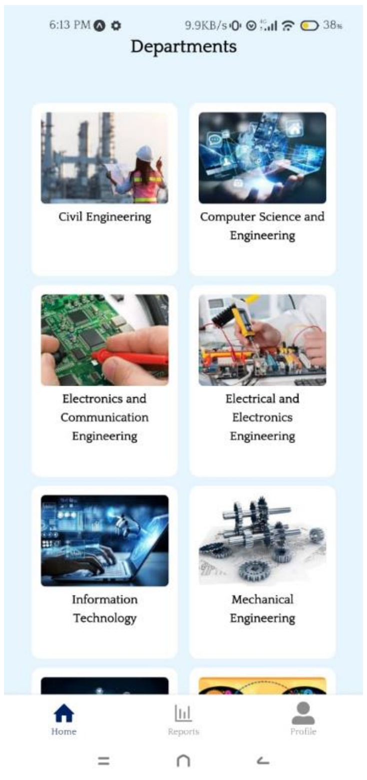
Output:

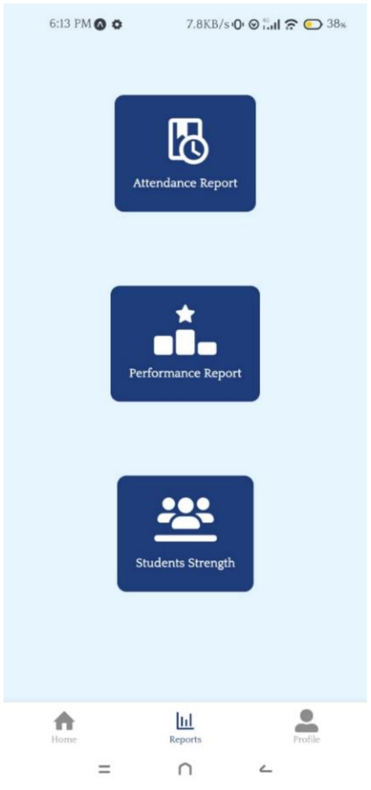


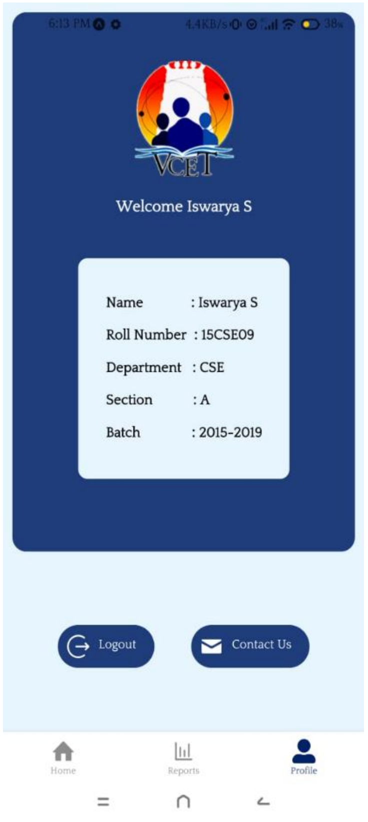


Invalid username or password. Please try again.

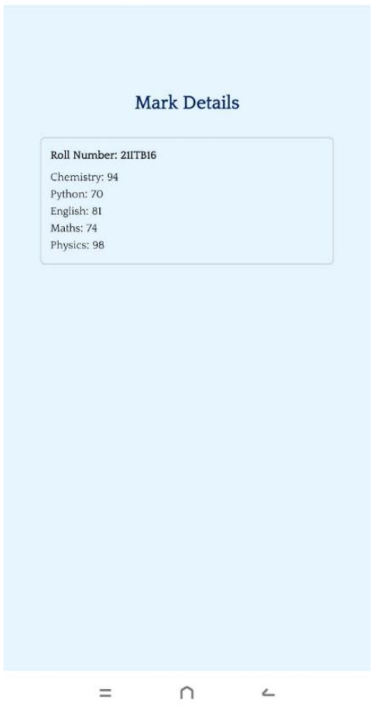

Velammal College of Engineering and Technology
Welcome ! Iswarya S (15CSE09)
Why VCET ?
Home
Reports
Profile


Departments
Civil Engineering
Computer Science and Engineering
Electronics and Communication Engineering
Electrical and Electronics Engineering
Information Technology
Mechanical Engineering
Home
Reports
Profile





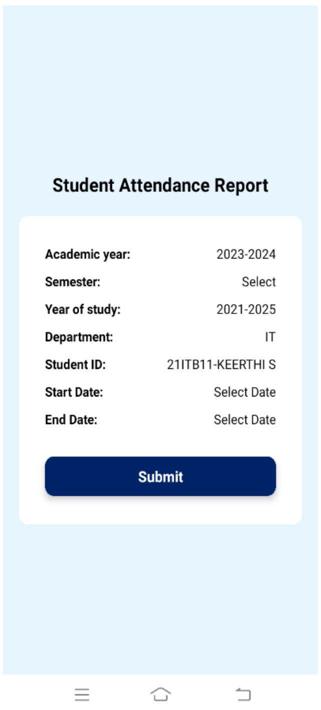
Name : Iswarya S
Roll Number : 15CSE09
Department : CSE
Section : A
Batch : 2015-2019



Mark Details

Roll Number: 21ITB16

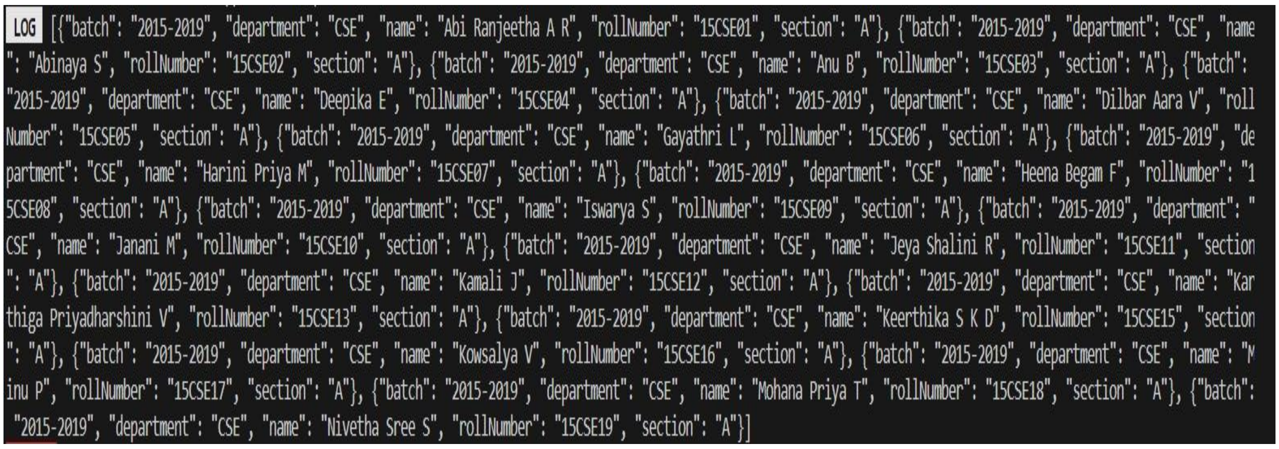
Chemistry: 94
Python: 70
English: 81
Maths: 74
Physics: 98



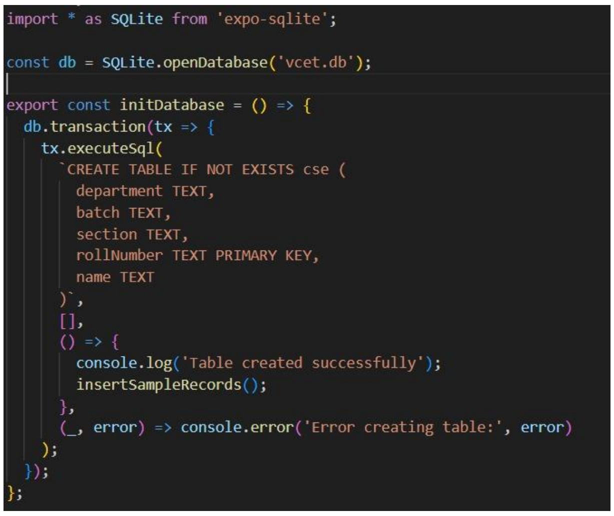
Student Attendance Report

Academic year:	2023-2024
Semester:	Select
Year of study:	2021-2025
Department:	IT
Student ID:	21ITB11-KEERTHI S
Start Date:	Select Date
End Date:	Select Date

BackEnd :



```
LOG [{"batch": "2015-2019", "department": "CSE", "name": "Abi Ranjeetha A R", "rollNumber": "15CSE01", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Abinaya S", "rollNumber": "15CSE02", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Anu B", "rollNumber": "15CSE03", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Deepika E", "rollNumber": "15CSE04", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Dilbar Aara V", "rollNumber": "15CSE05", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Gayathri L", "rollNumber": "15CSE06", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Harini Priya M", "rollNumber": "15CSE07", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Heena Begam F", "rollNumber": "15CSE08", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Iswarya S", "rollNumber": "15CSE09", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Janani M", "rollNumber": "15CSE10", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Jeya Shalini R", "rollNumber": "15CSE11", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Kamali J", "rollNumber": "15CSE12", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Karthiga Priyadharshini V", "rollNumber": "15CSE13", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Keerthika S K D", "rollNumber": "15CSE15", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Kowsalya V", "rollNumber": "15CSE16", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Minu P", "rollNumber": "15CSE17", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Mohana Priya T", "rollNumber": "15CSE18", "section": "A"}, {"batch": "2015-2019", "department": "CSE", "name": "Nivetha Sree S", "rollNumber": "15CSE19", "section": "A"}]
```



```
import * as SQLite from 'expo-sqlite';

const db = SQLite.openDatabase('vcet.db');

export const initDatabase = () => {
  db.transaction(tx => {
    tx.executeSql(
      `CREATE TABLE IF NOT EXISTS cse (
        department TEXT,
        batch TEXT,
        section TEXT,
        rollNumber TEXT PRIMARY KEY,
        name TEXT
      )`,
      [],
      () => {
        console.log('Table created successfully');
        insertSampleRecords();
      },
      (_, error) => console.error('Error creating table:', error)
    );
  });
};
```

Result :

In the Mini Project, An application “Student Information System “ with various features was developed, tested and the output was verified successfully.

Conclusion:

The development of the "Student Information System" app has successfully achieved its primary objectives of enhancing accessibility to academic information, promoting accountability among students, and improving overall user experience. The integration of features such as the login screen for secure authentication, performance tracking, and profile management has significantly contributed to a more connected and informed learning environment.

User feedback and testing results have indicated positive outcomes in terms of usability, performance, and user satisfaction. The app's use of modern technologies like React Native, Expo, and SQLite has proven to be effective in delivering a robust and responsive platform accessible across different devices and platforms.

Moving forward, the addition of staff login features to update data and generate performance reports represents an exciting avenue for future work. By empowering staff members with tools to analyze academic performance data, the app can further enhance its capabilities in facilitating data-driven decision-making and improving educational outcomes.

Future Work

The future work for the "Student Information System" app includes the development of staff login features aimed at updating data, analyzing academic performance metrics, and generating comprehensive reports. This will enable staff members to have real-time access to critical data, allowing them to identify trends, track student progress, and make informed decisions to improve the overall academic experience.

Additionally, incorporating data visualization tools and advanced analytics capabilities will further enhance the app's ability to provide actionable insights and support data-driven decision-making processes. Continuous enhancements to the user interface, performance optimization, and scalability improvements will also be prioritized to ensure the app remains efficient and user-friendly as it scales to accommodate a growing user base.

Collaboration with educational institutions and stakeholders will be key to gathering feedback, identifying evolving needs, and implementing new features and functionalities that align with industry best practices and standards. Overall, the future work aims to evolve the "Student Information System" app into a comprehensive and indispensable tool for enhancing educational management and improving student outcomes.