Ex. No: 1	Using react native, build a cross platform application for BMI calculator
Date:	

AIM:

To develop BMI calculator Using react native and cross platform application.

PROCEDURE:

Step 1: Set Up Your Development Environment

- Install Node.js and npm (Node Package Manager).
- Install React Native CLI globally using npm: npm install -g react-native-cli.
- Set up your development environment according to React Native's official documentation for your specific operating system.

Step 2: Create a New React Native Project

- Open your terminal or command prompt.
- Run react-native init BMICalculator to create a new React Native project called "BMICalculator".

Step 3: Design the User Interface

- Navigate into your project directory: cd BMICalculator.
- Open the project in your preferred code editor.
- Design the user interface using React Native components like View, Text, TextInput, Button, etc., to create input fields and buttons for weight and height input, as well as displaying the calculated BMI.

Step 4: Implement BMI Calculation Logic

- Write the logic to calculate BMI based on the input weight (in kilograms) and height (in meters).
- Create a function that takes weight and height as input and returns the calculated BMI.
- Use this function to calculate BMI whenever the user inputs their weight or height.

Step 5: Handle User Input

- Implement event handlers for input fields to capture user input for weight and height.
- Update the state of the component with the new input values.
- Trigger the BMI calculation function whenever there's a change in weight or height input.

Step 6: Display the Result

- Render the calculated BMI on the screen.
- Optionally, provide feedback to the user based on their BMI, such as displaying whether they are underweight, normal weight, overweight, or obese.

Step 7: Test Your Application

- Run your application on an emulator/simulator or a physical device to test its functionality.
- Verify that the BMI calculation and display work as expected.
- Test different input scenarios to ensure the robustness of your application.

Step 8: Styling and UI Enhancement

- Apply styles to make your UI visually appealing and user-friendly.
- Add animations or transitions to enhance the user experience.
- Consider accessibility aspects to ensure your app is usable by all users.

Step 9: Publish Your Application (Optional)

- If you wish to publish your application, follow the deployment guides for iOS and Android provided by React Native.
- Prepare your app for release, including setting up app icons, splash screens, and configuring build settings.
- Test your app thoroughly on real devices before publishing to app stores.

Step 10: Continuous Improvement and Maintenance

- Gather user feedback and iterate on your application to address any issues or feature requests.
- Keep your dependencies updated to ensure compatibility and security.
- Monitor your app's performance and address any performance issues that arise.

PROGRAM:

```
// App.js
import React, { useState } from 'react';
import { View, Text, TextInput, Button, StyleSheet } from 'react-native';
const App = () => {
  const [height, setHeight] = useState('');
  const [weight, setWeight] = useState('');
  const [bmi, setBMI] = useState(null);
  const calculateBMI = () => {
    const heightMeters = height / 100;
    const bmiValue = weight / (heightMeters * heightMeters);
    setBMI(bmiValue.toFixed(2));
  };
  return (
    <View style={styles.container}>
      <Text style={styles.title}>BMI Calculator</Text>
      <View style={styles.inputContainer}>
        <Text>Height (cm)</Text>
        <TextInput
```

```
          style={styles.input}
          value={height}
          onChangeText={setHeight}
          keyboardType="numeric"
        />
      </View>
      <View style={styles.inputContainer}>
        <Text>Weight (kg)</Text>
        <TextInput
          style={styles.input}
          value={weight}
          onChangeText={setWeight}
          keyboardType="numeric"
        />
      </View>
      <Button title="Calculate BMI" onPress={calculateBMI} />
      {bmi && <Text style={styles.result}>Your BMI: {bmi}</Text>}
    </View>
  );
};
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    padding: 20,
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 20,
  },
  inputContainer: {
    marginBottom: 10,
  },
  input: {
    borderWidth: 1,
    borderColor: '#ccc',
    borderRadius: 5,
    padding: 8,
    width: 200,
  },
  result: {
    marginTop: 20,
```

```
    fontSize: 18,
    fontWeight: 'bold',
  },
});
export default App;
```

index.js

```
import { AppRegistry } from "react-native";
import App from "./App";
AppRegistry.registerComponent("App", () => App);
AppRegistry.runApplication("App", {
  rootTag: document.getElementById("root")
});
```

OUTPUT:

Result:

Thus, the BMI calculator Android application was created and run successfully.

Ex.No: 2	Build a cross platform application for a simple expense manager which allows entering expenses and income on each day and displays category wise weekly income and expense
Date:	

AIM:

To. Build a cross platform application for a simple expense manager which allows entering expenses and income on each day and displays category wise weekly income and expense.

PROCEDURE:

Step 1: Set Up Your Development Environment

- Install Node.js and npm (Node Package Manager).
- Install React Native CLI globally using npm: npm install -g react-native-cli.
- Set up your development environment according to React Native's official documentation for your specific operating system.

Step 2: Create a New React Native Project

- Open your terminal or command prompt.
- Run react-native init ExpenseManager to create a new React Native project called "ExpenseManager".

Step 3: Design the User Interface

- Navigate into your project directory: cd ExpenseManager.
- Open the project in your preferred code editor.
- Design the user interface using React Native components like View, Text, TextInput, Button, etc., to create input fields for entering expenses and income, and display areas for showing weekly income and expense summaries by category.

Step 4: Set Up Data Management

- Decide on the data structure for storing expenses and income. You can use arrays, objects, or a database depending on the complexity of your application.
- Set up state management using React's built-in useState hook or a state management library like Redux to manage the application's state.

Step 5: Implement Expense and Income Entry

- Create input fields for entering expenses and income for each day.
- Implement logic to capture user input and update the state with the entered data.

- Store the entered expenses and income data in your chosen data structure.

Step 6: Calculate Weekly Summaries

- Write logic to calculate weekly income and expense summaries by category.
- Group expenses and income by category and calculate totals for each category.
- Display the weekly summaries on the UI.

Step 7: Handle Data Persistence

- Decide on the method of data persistence, such as using AsyncStorage for local storage or integrating with a backend server for cloud storage.
- Implement logic to save and retrieve expenses and income data from the chosen storage solution.

Step 8: Styling and UI Enhancement

- Apply styles to make your UI visually appealing and user-friendly.
- Add animations or transitions to enhance the user experience.
- Consider accessibility aspects to ensure your app is usable by all users.

Step 9: Test Your Application

- Run your application on an emulator/simulator or a physical device to test its functionality.
- Verify that expense and income entry, as well as weekly summaries, work as expected.
- Test different scenarios to ensure the robustness of your application.

Step 10: Publish Your Application

- If you wish to publish your application, follow the deployment guides for iOS and Android provided by React Native.
- Prepare your app for release, including setting up app icons, splash screens, and configuring build settings.
- Test your app thoroughly on real devices before publishing to app stores.

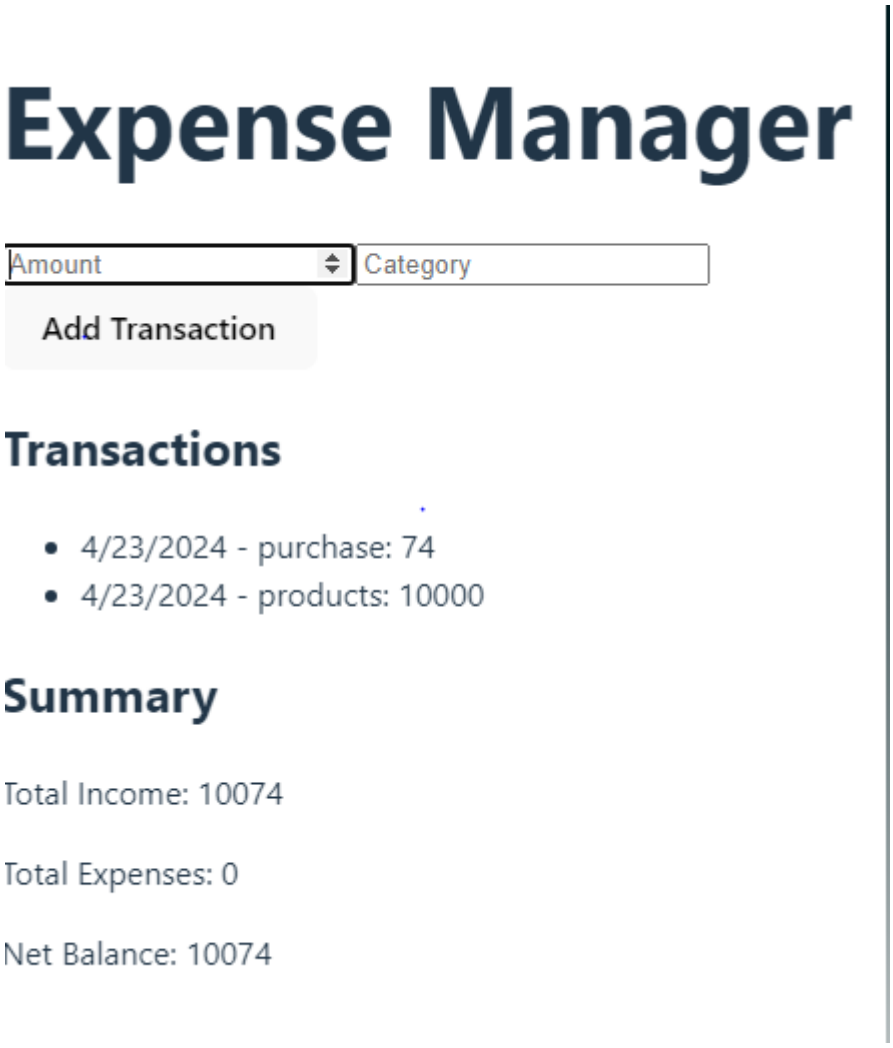
PROGRAM:

```
import React, { useState } from 'react';
function App() {
  const [transactions, setTransactions] = useState([]);
  const [amount, setAmount] = useState('');
  const [category, setCategory] = useState('');
  const addTransaction = () => {
    if (!amount || !category) return;
    const newTransaction = {
      amount: parseFloat(amount),
      category,
      date: new Date().toLocaleDateString(),
```

```
  };
  setTransactions([...transactions, newTransaction]);
  setAmount('');
  setCategory('');
 };
 const totalExpenses = transactions.reduce((acc, transaction) => {
  if (transaction.amount < 0) {
   return acc + transaction.amount;
  }
  return acc;
 }, 0);
 const totalIncome = transactions.reduce((acc, transaction) => {
  if (transaction.amount > 0) {
   return acc + transaction.amount;
  }
  return acc;
 }, 0);
 return (
  <div>
   <h1>Expense Manager</h1>
   <div>
    <input
     type="number"
     placeholder="Amount"
     value={amount}
     onChange={(e) => setAmount(e.target.value)}
    />
    <input
     type="text"
     placeholder="Category"
     value={category}
     onChange={(e) => setCategory(e.target.value)}
    />
    <button onClick={addTransaction}>Add Transaction</button>
   </div>
   <div>
    <h2>Transactions</h2>
    <ul>
     {transactions.map((transaction, index) => (
      <li key={index}>
       <span>{transaction.date}</span> - <span>{transaction.category}</span>:{' '}
       <span>{transaction.amount}</span>
      </li>
     ))}
```

```
          </ul>
        </div>
        <div>
          <h2>Summary</h2>
          <p>Total Income: {totalIncome}</p>
          <p>Total Expenses: {totalExpenses}</p>
          <p>Net Balance: {totalIncome + totalExpenses}</p>
        </div>
      </div>
    );
  }

  export default App;
```

OUTPUT:



RESULT:

Thus the cross platform application was created and run successfully.

Ex.No: 3	Develop a cross platform application to convert units from imperial system to metric system
Date:	

AIM:

To Develop a cross platform application to convert units from imperial system to metric system.

PROCEDURE:

Step 1: Set Up Your Development Environment

- Install Node.js and npm (Node Package Manager).
- Install React Native CLI globally using npm: npm install -g react-native-cli.
- Set up your development environment according to React Native's official documentation for your specific operating system.

Step 2: Create a New React Native Project

- Open your terminal or command prompt.
- Run react-native init UnitConverter to create a new React Native project called "UnitConverter".

Step 3: Design the User Interface

- Navigate into your project directory: cd UnitConverter.
- Open the project in your preferred code editor.
- Design the user interface using React Native components like View, Text, TextInput, Picker, Button, etc., to create input fields for entering values and selecting units, and display areas for showing the converted values.

Step 4: Implement Unit Conversion Logic

- Write logic to convert units from the imperial system to the metric system and vice versa.
- Create functions for each type of conversion (e.g., length, weight, volume) based on conversion formulas.
- Use these functions to convert units based on user input.

Step 5: Handle User Input

- Implement event handlers for input fields to capture user input for values.
- Update the state of the component with the new input values.
- Trigger the unit conversion functions whenever there's a change in input values or selected units.

Step 6: Display the Result

- Render the converted values on the screen.
- Display the converted values in a readable format.
- Optionally, provide feedback to the user based on the converted values.

Step 7: Test Your Application

- Run your application on an emulator/simulator or a physical device to test its functionality.
- Verify that unit conversion works correctly for different types of units (e.g., length, weight, volume).
- Test different input scenarios to ensure the accuracy of your conversion logic.

Step 8: Styling and UI Enhancement

- Apply styles to make your UI visually appealing and user-friendly.
- Add animations or transitions to enhance the user experience.
- Consider accessibility aspects to ensure your app is usable by all users.

Step 9: Publish Your Application (Optional)

- If you wish to publish your application, follow the deployment guides for iOS and Android provided by React Native.
- Prepare your app for release, including setting up app icons, splash screens, and configuring build settings.
- Test your app thoroughly on real devices before publishing to app stores.

Step 10: Continuous Improvement and Maintenance

- Gather user feedback and iterate on your application to address any issues or feature requests.
- Keep your dependencies updated to ensure compatibility and security.
- Monitor your app's performance and address any performance issues that arise.

PROGRAM:

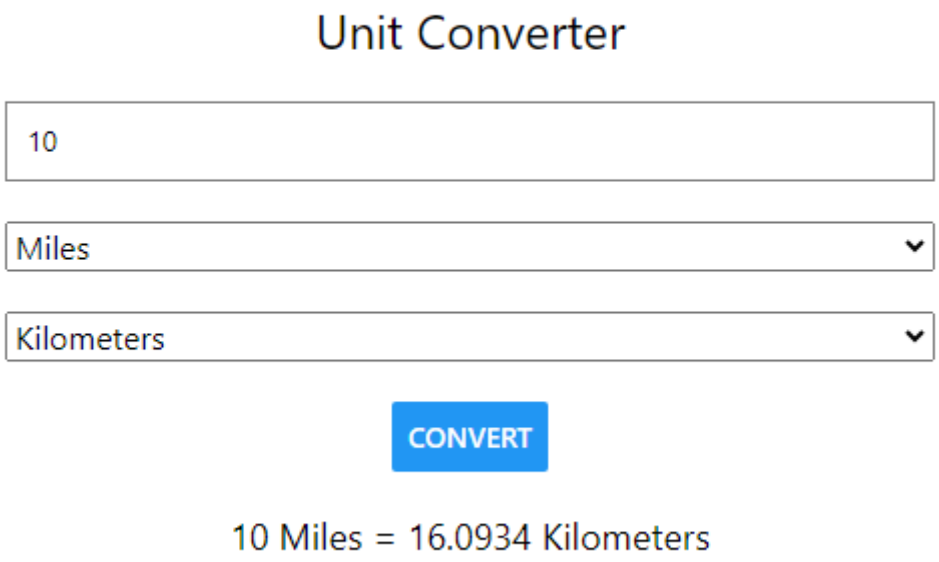
```
import React, { useState } from 'react';
import { StyleSheet, Text, View, TextInput, Button, Picker } from 'react-native';
export default function App() {
  const [inputValue, setInputValue] = useState('');
  const [convertedValue, setConvertedValue] = useState('');
  const [inputUnit, setInputUnit] = useState('Kilometers');
  const [convertedUnit, setConvertedUnit] = useState('Miles');
  const [resultText, setResultText] = useState('');
```

```
const convertUnit = () => {
  let converted = 0;
  switch (inputUnit) {
    case 'Kilometers':
      converted = convertedUnit === 'Miles' ? parseFloat(inputValue) * 0.621371 : 0;
      break;
    case 'Miles':
      converted = convertedUnit === 'Kilometers' ? parseFloat(inputValue) * 1.60934 : 0;
      break;
    case 'Kilograms':
      converted = convertedUnit === 'Pounds' ? parseFloat(inputValue) * 2.20462 : 0;
      break;
    case 'Pounds':
      converted = convertedUnit === 'Kilograms' ? parseFloat(inputValue) * 0.453592 : 0;
      break;
    default:
      setResultText('Unsupported conversion');
      return;
  }
  setResultText(`${inputValue} ${inputUnit} = ${converted} ${convertedUnit}`);
};
return (
  <View style={styles.container}>
    <Text style={styles.title}>Unit Converter</Text>
    <TextInput
      style={styles.input}
      keyboardType="numeric"
      placeholder="Enter value"
      value={inputValue}
```

```
        onChangeText={(text) => setInputValue(text)}
      />
      <Picker
        selectedValue={inputUnit}
        style={styles.picker}
        onValueChange={(itemValue) => setInputUnit(itemValue)}>
        <Picker.Item label="Kilometers" value="Kilometers" />
        <Picker.Item label="Miles" value="Miles" />
        <Picker.Item label="Kilograms" value="Kilograms" />
        <Picker.Item label="Pounds" value="Pounds" />
      </Picker>
      <Picker
        selectedValue={convertedUnit}
        style={styles.picker}
        onValueChange={(itemValue) => setConvertedUnit(itemValue)}>
        <Picker.Item label="Miles" value="Miles" />
        <Picker.Item label="Kilometers" value="Kilometers" />
        <Picker.Item label="Pounds" value="Pounds" />
        <Picker.Item label="Kilograms" value="Kilograms" />
      </Picker>
      <Button title="Convert" onPress={convertUnit} />
      <Text style={styles.result}>{resultText}</Text>
    </View>
  );
}
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
```

```
    alignItems: 'center',
    justifyContent: 'center',
  },
  title: {
    fontSize: 24,
    marginBottom: 20,
  },
  input: {
    width: '80%',
    height: 40,
    borderColor: 'gray',
    borderWidth: 1,
    marginBottom: 20,
    paddingHorizontal: 10,
  },
  picker: {
    width: '80%',
    marginBottom: 20,
  },
  result: {
    marginTop: 20,
    fontSize: 18,
  },
});
```

OUTPUT:



RESULT:

Thus the cross platform application to convert units from imperial system to metric system program is implemented successfully.

Ex.No: 4	Design and develop a cross platform application for day to day task(to-do) management
Date:	

AIM:

To run and execute cross platform application for day to day task (to-do) management.

PROCEDURE:

Step 1: Define Requirements and Features

- Identify the core features your to-do list app will include, such as adding tasks, marking tasks as complete, setting deadlines, categorizing tasks, etc.
- Determine the user experience flow, including navigation between screens and interactions with tasks.

Step 2: Set Up Your Development Environment

- Install Node.js and npm (Node Package Manager).
- Install React Native CLI globally using npm: npm install -g react-native-cli.
- Set up your development environment according to React Native's official documentation for your specific operating system.

Step 3: Create a New React Native Project

- Open your terminal or command prompt.
- Run react-native init ToDoList to create a new React Native project called "ToDoList".

Step 4: Design the User Interface

- Use wireframing tools or pen and paper to sketch out the UI layout for your to-do list app.
- Design the user interface using React Native components like View, Text, TextInput, Button, FlatList, etc., to create screens for displaying tasks, adding tasks, and task details.

Step 5: Implement Task Management Logic

- Set up data management using state management libraries like Redux or React Context API.
- Create functions to add tasks, mark tasks as complete, delete tasks, edit tasks, and categorize tasks.
- Implement logic to store tasks locally on the device or sync tasks with a backend server for cloud storage.

Step 6: Handle User Input

- Implement event handlers for adding, editing, and deleting tasks.

- Capture user input for task details such as title, description, deadline, priority, etc.
- Validate user input to ensure data integrity and usability.

Step 7: Display Tasks

- Render the list of tasks on the main screen using a FlatList or similar component.
- Display task details such as title, description, deadline, priority, and completion status.
- Implement sorting and filtering options to organize tasks based on priority, deadline, etc.

Step 8: Enhance User Experience

- Implement features such as drag-and-drop for reordering tasks, swipe actions for quick task management, and gestures for navigation.
- Add animations or transitions to provide visual feedback and enhance the user experience.
- Ensure responsiveness across different screen sizes and orientations.

Step 9: Test Your Application

- Run your application on an emulator/simulator or a physical device to test its functionality.
- Test adding, editing, deleting, and completing tasks to ensure they work as expected.
- Test edge cases and error scenarios to ensure the robustness of your application.

Step 10: Styling and UI Polish

- Apply styles to make your UI visually appealing and consistent with platform guidelines.
- Use icons, colors, and typography to enhance readability and usability.
- Test your app's accessibility and ensure it's usable by users with disabilities.

Step 11: Publish Your Application (Optional)

- If you wish to publish your application, follow the deployment guides for iOS and Android provided by React Native.
- Prepare your app for release, including setting up app icons, splash screens, and configuring build settings.
- Test your app thoroughly on real devices before publishing to app stores.

Step 12: Continuous Improvement and Maintenance

- Gather user feedback and iterate on your application to address any issues or feature requests.
- Keep your dependencies updated to ensure compatibility and security.
- Monitor your app's performance and address any performance issues that arise.

PROGRAM:

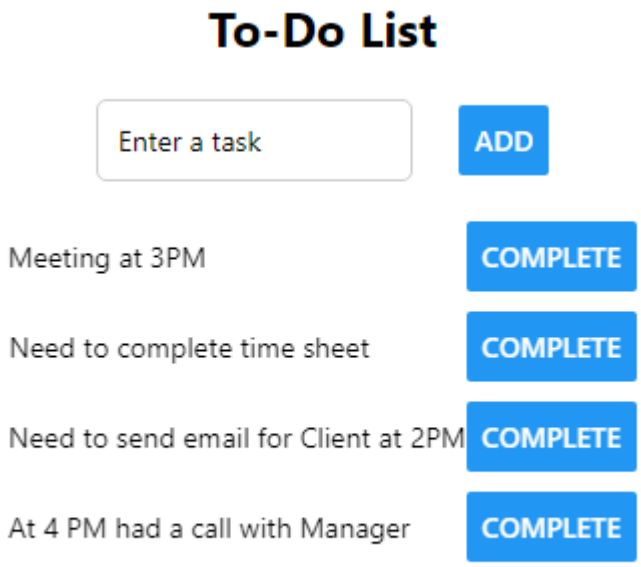
```
import React, { useState } from 'react';

import { StyleSheet, Text, View, TextInput, Button, FlatList } from 'react-native';
```

```
export default function App() {
  const [task, setTask] = useState('');
  const [tasks, setTasks] = useState([]);
  const addTask = () => {
    if (task.trim() !== '') {
      setTasks([...tasks, { id: Math.random().toString(), text: task }]);
      setTask('');
    }
  };
  const completeTask = (taskId) => {
    setTasks(tasks.filter((task) => task.id !== taskId));
  };
  return (
    <View style={styles.container}>
      <Text style={styles.heading}>To-Do List</Text>
      <View style={styles.inputContainer}>
        <TextInput
          style={styles.input}
          placeholder="Enter a task"
          onChangeText={(text) => setTask(text)}
          value={task}
        />
        <Button title="Add" onPress={addTask} />
      </View>
      <FlatList
        data={tasks}
        keyExtractor={(item) => item.id}
```

```
        renderItem={({ item }) => (
          <View style={styles.task}>
            <Text>{item.text}</Text>
            <Button title="Complete" onPress={() => completeTask(item.id)} />
          </View>
        )}
      />
    </View>
  );
}
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
    padding: 20,
  },
  heading: {
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 20,
  },
  inputContainer: {
    flexDirection: 'row',
    justifyContent: 'space-between',
    alignItems: 'center',
```

```
    marginBottom: 20,
  },
  input: {
    width: '70%',
    borderWidth: 1,
    borderColor: '#ccc',
    padding: 10,
    borderRadius: 5,
  },
  task: {
    flexDirection: 'row',
    justifyContent: 'space-between',
    alignItems: 'center',
    marginBottom: 10,
    width: '100%',
  },
});
```

OUTPUT:



RESULT:

Thus the To Do Application was developed successfully using cross platform.

Ex.No: 5	Design an android application using cordova for a user login screen with username, password, reset button and a submit button. Also include header image and a label. Use layout managers
Date:	

AIM:

To run and execute android application using cordova for a user login screen with username, password, reset button and a submit button. Also include header image and a label. Use layout managers.

PROCEDURE:

Step 1: Set Up Your Cordova Project

- Install Cordova globally via npm if you haven't already: npm install -g cordova.
- Create a new Cordova project by running: cordova create LoginApp.
- Navigate into your project directory: cd LoginApp.

Step 2: Add Platforms

- Add platforms for which you want to build the app, for example:
- For Android: cordova platform add android
- For iOS: cordova platform add ios

Step 3: Design User Interface (HTML)

- Open www/index.html in your preferred text editor.

Step 4: Style User Interface (CSS)

- Create a new CSS file named www/css/index.css.
- Add your CSS styles to index.css to style the user interface according to your requirements.

Step 5: Implement JavaScript Functionality

- Open www/js/index.js in your text editor.

Step 6: Add Header Image

- Place your header image in the www/img directory.
- Ensure the src attribute of the <img> tag in index.html points to the correct path of your header image.

Step 7: Test Your Application

- Run your Cordova application on a simulator or device using cordova run android or cordova run ios.
- Test the user login screen to ensure that it functions as expected.

PROGRAM:

index.html

```
<!DOCTYPE html>
<html>
<head>
```

```
    <title>Login</title>
    <link rel="stylesheet" type="text/css" href="css/index.css">
</head>
<body>
    <div class="header">
        <img src="img/header_image.png" alt="Header Image">
        <h1>Login</h1>
    </div>
    <div class="login-container">
        <label for="username">Username</label>
        <input type="text" id="username" name="username">
        <label for="password">Password</label>
        <input type="password" id="password" name="password">
        <button id="resetBtn">Reset</button>
        <button id="submitBtn">Submit</button>
    </div>
    <script type="text/javascript" src="js/index.js"></script>
</body>
</html>
```

style.css

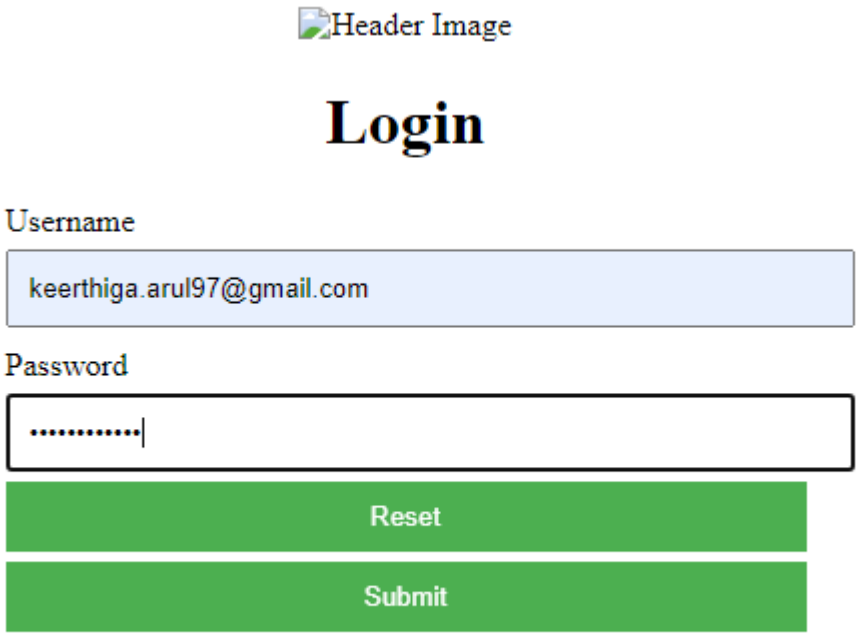
```
.header {
    text-align: center;
}
.login-container {
    margin: 0 auto;
    width: 80%;
    max-width: 400px;
}
label {
    display: block;
    margin-top: 10px;
}
input[type="text"],
input[type="password"],
button {
    width: 100%;
    padding: 10px;
    margin-top: 5px;
button {
    background-color: #4CAF50;
    color: white;
    border: none;
    cursor: pointer;
```

```
}
button:hover {
   opacity: 0.8;
}
```

script.js

```
document.addEventListener('deviceready', onDeviceReady, false);
function onDeviceReady() {
   document.getElementById('resetBtn').addEventListener('click', resetForm);
   document.getElementById('submitBtn').addEventListener('click', submitForm);
}
function resetForm() {
   document.getElementById('username').value = '';
   document.getElementById('password').value = '';
}
function submitForm() {
   var username = document.getElementById('username').value;
   var password = document.getElementById('password').value;
   // Perform login validation here
   // Example: You can use AJAX to send the login credentials to the server
   console.log("Username: " + username + ", Password: " + password);
}
```

OUTPUT:



RESULT:

Thus, the application was created successfully.

Ex.No: 6	Design and develop a android application using Apache Cordova to find and display the current location of the user.
Date:	

AIM:

To Design and develop a android application using Apache Cordova to find and display the current location of the user.

PROCEDURE:

Step 1: Set Up Your Cordova Project

- Install Cordova globally via npm if you haven't already: npm install -g cordova.
- Create a new Cordova project by running: cordova create LocationApp.
- Navigate into your project directory: cd LocationApp.

Step 2: Add Android Platform

- Add the Android platform to your Cordova project: cordova platform add android.

Step 3: Install Required Plugins

- Install the Geolocation plugin for accessing the device's GPS location:
- cordova plugin add cordova-plugin-geolocation

Step 4: Design User Interface (HTML)

- Open www/index.html in your preferred text editor.

Step 5: Implement JavaScript Functionality

- Open www/js/index.js in your text editor.
- Add the following JavaScript function to retrieve the device's current location:

Step 6: Test Your Application

- Run your Cordova application on an Android emulator or device using: cordova run android.
- Click the "Get Location" button to retrieve and display the current location of the user.

PROGRAM:

INDEX.HTML

```
<!DOCTYPE html>

<html>
```

```
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=no">
    <title>Location App</title>
    <link rel="stylesheet" type="text/css" href="css/index.css" />
</head>
<body>
    <div class="container">
        <h1>Location App</h1>
        <button onclick="getLocation()">Get Location</button>
        <div id="location"></div>
    </div>
    <script type="text/javascript" src="script.js"></script>
    <script type="text/javascript" src="js/index.js"></script>
</body>
</html>
```

SCRIPT.JS

```
function getLocation() {
    navigator.geolocation.getCurrentPosition(
        onSuccess,
        onError,
        { enableHighAccuracy: true }
    );
}
function onSuccess(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    document.getElementById('location').innerHTML = 'Latitude: ' + latitude + '<br>Longitude: ' + longitude;
```

```
}
function onError(error) {
    alert('Error occurred: ' + error.message);
}
```

Result:

Thus, the android application using Apache Cordova to find and display the current location of the user is implemented successfully.

Ex.No: 7	Write programs using Java to create Android application having Databases
Date:	

AIM:

To Write programs using Java to create Android application having Databases

PROCEDURE:

Step 1: Set Up Your Android Project

- Open Android Studio and create a new Android project.
- Choose an appropriate project name, package name, and other settings.
- Select "Empty Activity" as the template for your project.

Step 2: Design the User Interface

- Open res/layout/activity_main.xml.
- Design the user interface to include input fields for adding books (e.g., title, author, ISBN), a button to add a book, and a RecyclerView to display the list of books.

Step 3: Set Up the Database

- Create a new Java class named Book.
- Define the attributes of the Book class (e.g., title, author, ISBN).
- Create a subclass of SQLiteOpenHelper to manage the database creation and version management.
- Define the database schema and create the necessary tables.

Step 4: Implement CRUD Operations

- Create methods in your database helper class to perform CRUD (Create, Read, Update, Delete) operations on the database.
- Implement methods to insert new books into the database, retrieve all books, update book details, and delete books.

Step 5: Implement Business Logic

- In your main activity (MainActivity.java), implement logic to interact with the database.
- Implement event handlers for adding books, updating the RecyclerView to display the list of books, etc.

Step 6: Display Data in RecyclerView

- Set up a RecyclerView in your main activity layout (activity_main.xml).

- Create a custom adapter class to bind data from the database to the RecyclerView.
- Populate the RecyclerView with data retrieved from the database.

Step 7: Test Your Application

- Run your application on an Android emulator or physical device.
- Test adding, updating, and deleting books to ensure they work as expected.
- Verify that data persists across app launches.

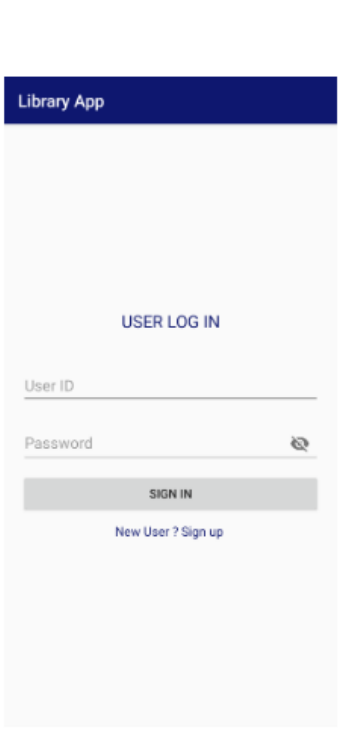
PROGRAM:

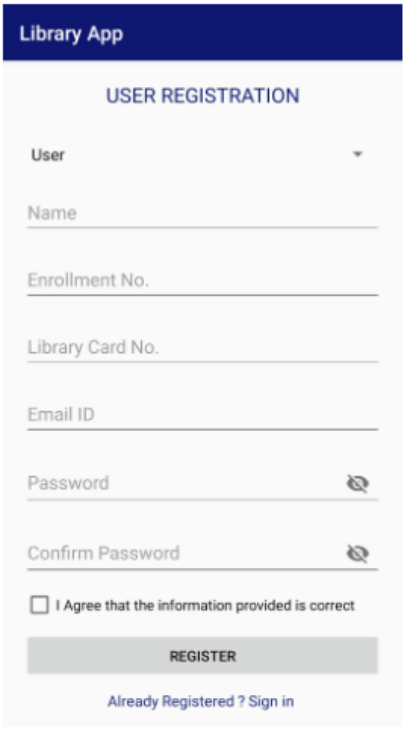
```
// Book.java
public class Book {
    private long id;
    private String title;
    private String author;
    private String isbn;
    // Getters and setters
}
// DatabaseHelper.java
public class DatabaseHelper extends SQLiteOpenHelper {
    private static final String DATABASE_NAME = "library.db";
    private static final int DATABASE_VERSION = 1;
    private static final String TABLE_BOOKS = "books";
    private static final String COLUMN_ID = "id";
    private static final String COLUMN_TITLE = "title";
    private static final String COLUMN_AUTHOR = "author";
    private static final String COLUMN_ISBN = "isbn";
    public DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    @Override
```

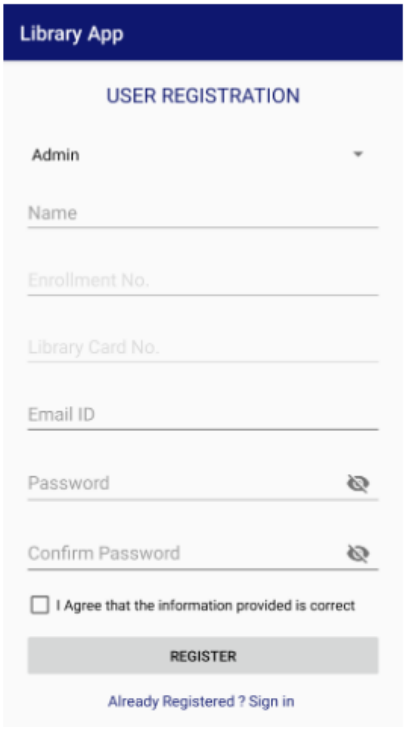
```
    public void onCreate(SQLiteDatabase db) {
        String createTableQuery = "CREATE TABLE " + TABLE_BOOKS + " (" +
                COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                COLUMN_TITLE + " TEXT, " +
                COLUMN_AUTHOR + " TEXT, " +
                COLUMN_ISBN + " TEXT" + ")";
        db.execSQL(createTableQuery);
    }
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_BOOKS);
        onCreate(db);
    }
    // CRUD operations methods
}
// MainActivity.java
public class MainActivity extends AppCompatActivity {
    private DatabaseHelper dbHelper;
    // Declare RecyclerView, adapter, and other variables

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main)
        dbHelper = new DatabaseHelper(this);
        // Initialize RecyclerView, adapter, and other variables
        // Set up event handlers
```

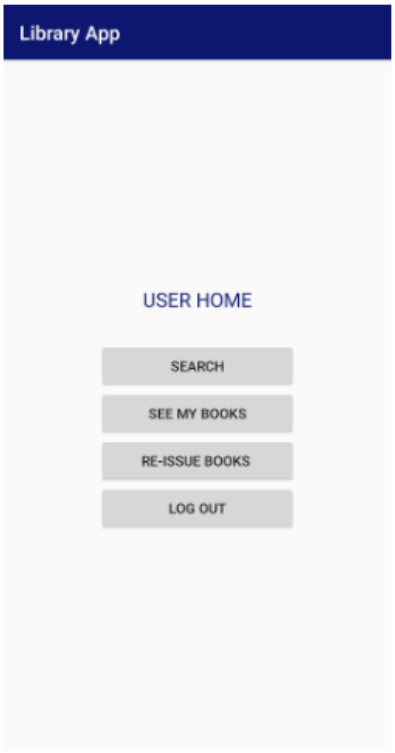
```
    }
    // Implement methods to interact with the database and update UI
}
```

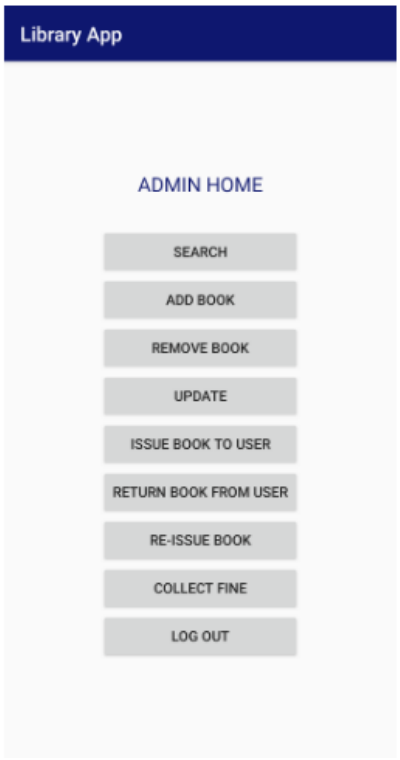
OUTPUT:

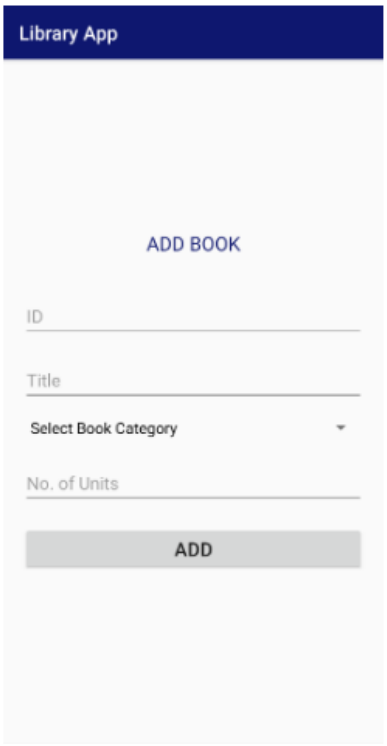












RESULT:

Thus, the android application using java to create library application program was implemented successfully.