Міністерство освіти і науки України
Чорноморський національний університет імені Петра Могили

Методична серія. Випуск 464

Заснована в 2016 році

Г. В. Горбань, В. С. Раленко

ПРОГРАМУВАННЯ МОБІЛЬНИХ ПРИСТРОЇВ

Методичні рекомендації до виконання лабораторних робіт

ЧНУ
ВИДАВНИЦТВО
Миколаїв – 2025

УДК 004.4:621.395.721.5](076.5)
Г67

Рекомендовано до друку Вченою радою факультету комп'ютерних наук Чорноморського національного університету імені Петра Могили (протокол № 9 від 10 квітня 2025 року).

Рецензент:
Євген СІДЕНКО, канд. техн. наук, доцент, доцент кафедри інтелектуальних інформаційних систем, ЧНУ імені Петра Могили.

Горбань Г. В.

Г67 Програмування мобільних пристроїв : метод. рек. до виконання лабораторних робіт / Г. В. Горбань, В. С. Раленко. – Миколаїв : Вид-во ЧНУ ім. Петра Могили, 2025. – 152 с. – (Методична серія ; вип. 464).

Методичні вказівки містять опис лабораторних робіт з дисципліни «Програмування мобільних пристроїв», які присвячені освоєнню засобів та методів розробки застосунків за допомогою фреймворку React Native для мобільних пристроїв, що працюють на операційних системах Android та iOS. У лабораторних роботах поетапно розглянуто налаштування середовища розробки, проєктування інтерфейсів користувача, створення багатовіконних (багатоекранних) застосунків та інтеграцію мультимедійних можливостей у мобільні застосунки. Методичні вказівки призначено для студентів спеціальностей F2 – «Інженерія програмного забезпечення» та F3 – «Комп'ютерні науки», а також можуть бути корисними для здобувачів вищої освіти інших спеціальностей галузі знань F – «Інформаційні технології».

УДК 004.4:621.395.721.5](076.5)

ISSN 1811-492X

ЗМІСТ

ВСТУП

Розробка нативних мобільних застосунків може бути складною. Зі складними середовищами, багатослівними фреймворками і тривалим часом компіляції, з яким стикаються розробники, розробка якісного нативного мобільного застосунку є непростим завданням. Не дивно, що на ринку з'явилися рішення, які намагаються вирішити проблеми, пов'язані з розробкою нативних мобільних застосунків, і намагаються зробити її простішою.

В основі цієї складності лежить перешкода крос-платформної розробки. Різні платформи принципово відрізняються одна від одної і не мають спільних середовищ розробки, API або коду. Через це ми повинні мати окремі команди, які працюють над кожною платформою, що є і дорогим, і неефективним.

Але зараз захоплюючий час у розробці мобільних застосунків. Ми є свідками нової парадигми в ландшафті мобільної розробки, і React Native знаходиться на передньому краї цієї зміни в тому, як ми створюємо і проєктуємо мобільні застосунки. Тепер можна створювати нативні крос-платформні застосунки, а також вебзастосунки, використовуючи одну мову та одну команду. З ростом кількості мобільних пристроїв і подальшим збільшенням попиту на талановитих розробників, що призводить до зростання зарплат розробників, React Native дає можливість створювати якісні застосунки на всіх платформах за менші витрати часу і коштів, забезпечуючи при цьому якісний користувальницький інтерфейс і чудовий досвід для розробників.

Основною метою вивчення дисципліни є отримання досвіду розробки програмного забезпечення для різних мобільних платформ із використанням сучасних інструментів. Освоєння дисципліни базується на концепціях системного аналізу та сучасних технологій розробки інформаційних систем.

У результаті вивчення дисципліни здобувачі повинні мати *знання* з:

– основних компонентів архітектури мобільних платформ;
– життєвого циклу мобільних застосунків та їхньої структури;
– основних компонентів React Native;
– основних елементів користувацького інтерфейсу мобільних застосунків;
– інструментів для програмування та основ проєктування мобільних застосунків.

У результаті вивчення дисципліни здобувачі повинні *уміти*:

– програмувати та проводити ефективне тестування програм і застосунків для мобільних пристроїв;

– практично застосовувати інструментальні засоби і методи розробки мобільних застосунків.

ЛАБОРАТОРНА РОБОТА №1 «ПОЧАТКОВЕ НАЛАШТУВАННЯ СЕРЕДОВИЩА ДЛЯ РОБОТИ З REACT NATIVE. СТВОРЕННЯ ТА ЗАПУСК ПЕРШОГО ЗАСТОСУНКУ REACT NATIVE»

Теоретичні відомості

React Native дозволяє створювати мобільні застосунки, використовуючи тільки JavaScript. Він використовує той же дизайн, що і React, дозволяючи створювати багатий мобільний інтерфейс із декларативних компонентів. З React Native ви не створюєте мобільний вебзастосунок, HTML5-застосунок або гібридний застосунок; ви створюєте справжній мобільний застосунок, який неможливо відрізнити від застосунка, створеного за допомогою Objective-C або Java. React Native використовує ті ж самі фундаментальні будівельні блоки інтерфейсу, що і звичайні застосунки для iOS та Android. Ви просто збираєте ці будівельні блоки разом за допомогою JavaScript та React.

Особливості React Native

– React – це фреймворк для створення веб та мобільних застосунків за допомогою JavaScript;

– Нативний – Ви можете використовувати нативні компоненти, керовані JavaScript;

– Платформи – React Native підтримує платформи IOS та Android.

Переваги React Native

– JavaScript – Ви можете використовувати існуючі знання JavaScript для створення нативних мобільних застосунків.

– Спільний доступ до коду – Ви можете ділитися більшою частиною свого коду на різних платформах.

– Спільнота – Спільнота навколо React та React Native велика, і ви зможете знайти будь-яку відповідь, яка вам потрібна.

Обмеження React Native

Нативні компоненти – якщо ви хочете створити нативну функціональність, яка ще не створена, вам потрібно буде написати деякий специфічний для платформи код. [1]

Налаштування середовища розробки

Якщо ви новачок у мобільній розробці, найпростіший спосіб почати – це Expo Go. Expo – це набір інструментів і сервісів, побудованих навколо React Native, і, хоча він має багато функцій, найбільш актуальною для нас зараз є те, що він може допомогти вам написати застосунок на React Native за лічені хвилини. Вам знадобиться лише остання версія Node.js та телефон або емулятор. Якщо ви хочете спробувати React Native безпосередньо у вашому веб-браузері, перш ніж встановлювати будь-які інструменти, ви можете спробувати Snack (https://snack.expo.dev/).

Якщо ви вже знайомі з мобільною розробкою, ви можете використовувати React Native CLI. Для початку вам знадобиться Xcode або Android Studio. Якщо у вас вже встановлений один з цих інструментів, ви зможете почати роботу протягом декількох хвилин. Якщо вони не встановлені, вам слід очікувати, що ви витратите близько години на їх встановлення та налаштування.

Expo CLI та Expo Go

Для розробки застосунків за допомогою Expo вам знадобляться два інструменти. Інструмент командного рядка під назвою Expo CLI для обслуговування вашого проєкту та мобільний клієнтський застосунок під назвою Expo Go для відкриття проєкту на платформах Android та iOS. Крім того, ви можете використовувати будь-який веб-браузер для запуску проєкту в Інтернеті. [2]

Для використання Expo CLI на машині розробника повинні бути встановлені наступні інструменти:

– Випуск Node.js LTS – рекомендується використовувати тільки випуски Node.js LTS (з парними номерами). Як офіційно заявляє Node.js, «Виробничі програми повинні використовувати тільки випуски Active LTS або Maintenance LTS»;
– Git;
– Для користувачів macOS або Linux: Watchman.

Рекомендовані інструменти

– Yarn
– VS Code Editor
– Розширення VS Code Expo для налагодження конфігурації Expo та автозаповнення.

Для користувачів Windows рекомендується використовувати PowerShell, Bash через WSL або термінал VS Code.

Використання Expo CLI

Ви можете використовувати Expo CLI без установки, використовуючи **npx** – запуск пакетів Node.js.

Наприклад, щоб побачити список доступних команд в Expo CLI, відкрийте термінал на вашій машині розробки і виконайте наступну команду:

npx expo -h

Для встановлення Expo CLI слід використовувати команду:

npm install -g expo-cli

Створення нового застосунку

Перш ніж створювати новий застосунок Expo, ви повинні переконатися, що:

– Expo CLI працює на вашій машині для розробки

– застосунок Expo Go встановлено на вашому фізичному пристрої або емуляторі/симуляторі Android або iOS

Щоб ініціалізувати новий проєкт, потрібно використати команду **create-expo-app**:

npx create-expo-app my-app,

а потім перейти до папки створеного проєкту:

cd my-app

Ви також можете використовувати опцію --template з командою create-expo-app. Потрібно запустити npx create-expo-app --template, щоб побачити список доступних шаблонів. (рис. 1.1)

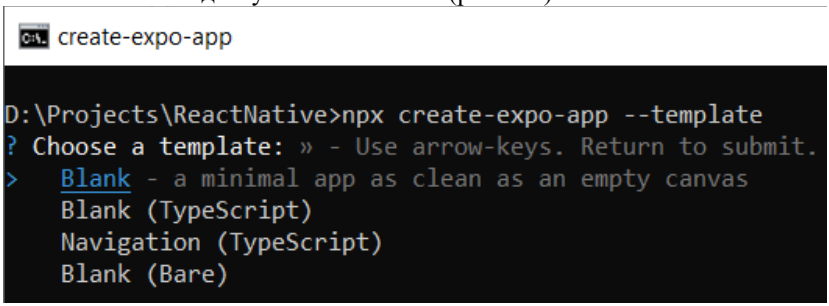


Рисунок 1.1 – Створення проєкту

За замовчуванням буде створений порожній (Blank) проєкт.

Запуск сервера розробки

Запустити сервер розробки можна, виконавши наступну команду:
npx expo start

Коли ми запускаємо npx expo start (або yarn expo start), Expo CLI запускає Metro Bundler. Це HTTP-сервер, який компілює JavaScript-код нашого застосунку за допомогою компілятора Babel і подає його застосунку Expo. (рис. 1.2)

Рисунок 1.2 – Запуск сервера розробки Expo

Щоб відкрити застосунок:

– на пристрої Android натисніть «Відсканувати QR-код» на вкладці «Головна» застосунку Expo Go та відскануйте QR-код, який ви побачите на терміналі.

– на вашому iPhone або iPad відкрийте стандартний застосунок Apple «Камера» та відскануйте QR-код, який ви побачите в терміналі.

Ви можете відкрити проєкт на декількох пристроях одночасно.

По-перше, потрібно переконайтесь, що ви перебуваєте в одній мережі Wi-Fi на вашому комп'ютері та пристрої.

Якщо це все ще не працює, це може бути пов'язано з конфігурацією маршрутизатора – це типово для публічних мереж. Ви можете обійти це, вибравши тип з'єднання «Тунель» при запуску сервера розробки, а потім відсканувавши QR-код ще раз.

npx expo start --tunnel

Використання типу з'єднання «Тунель» робить перезавантаження програми значно повільнішим, ніж в режимі «LAN» або «Локальний», тому краще уникати тунельного з'єднання, коли це можливо. Ви можете встановити симулятор/емулятор для прискорення розробки, якщо «Тунель» необхідний для доступу до вашої машини з іншого пристрою у вашій мережі.

Якщо ви використовуєте симулятор або емулятор, ви можете знайти наступні комбінації клавіш Expo CLI корисними для відкриття програми на будь-якій з наступних платформ:

– натискання **a** відкриє в емуляторі Android або на підключеному пристрої;

– натискання **i** відкриє в емуляторі iOS;

– натискання **w** відкриє застосунок у веб-браузері. Expo підтримує всі основні браузери.

Встановлення Android Studio

Відвідайте веб-сторінку https://developer.android.com/studio/ та завантажте Android studio.

Далі після встановлення, запуску програми та створення нового проєкту слід налаштувати віртуальний пристрій. Для цього спочатку треба встановити SDK. Для цього у пункті меню **Tools** обираємо **SDK Manager**. (рис. 1.3)

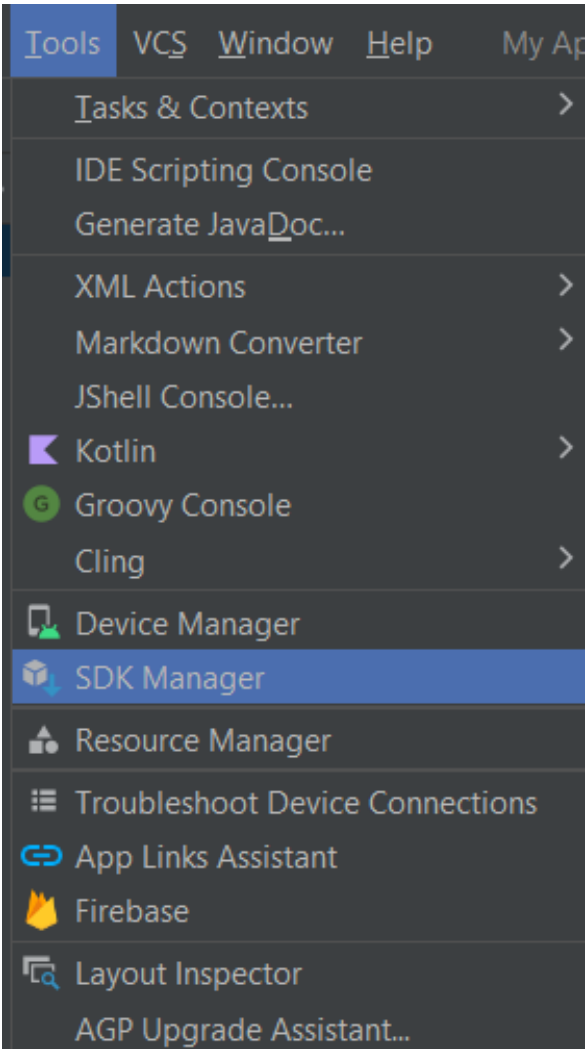


Рисунок 1.3 – Вибір SDK Manager

Далі обираємо необхідну версію ОС Android для встановлення: (рис. 1.4)

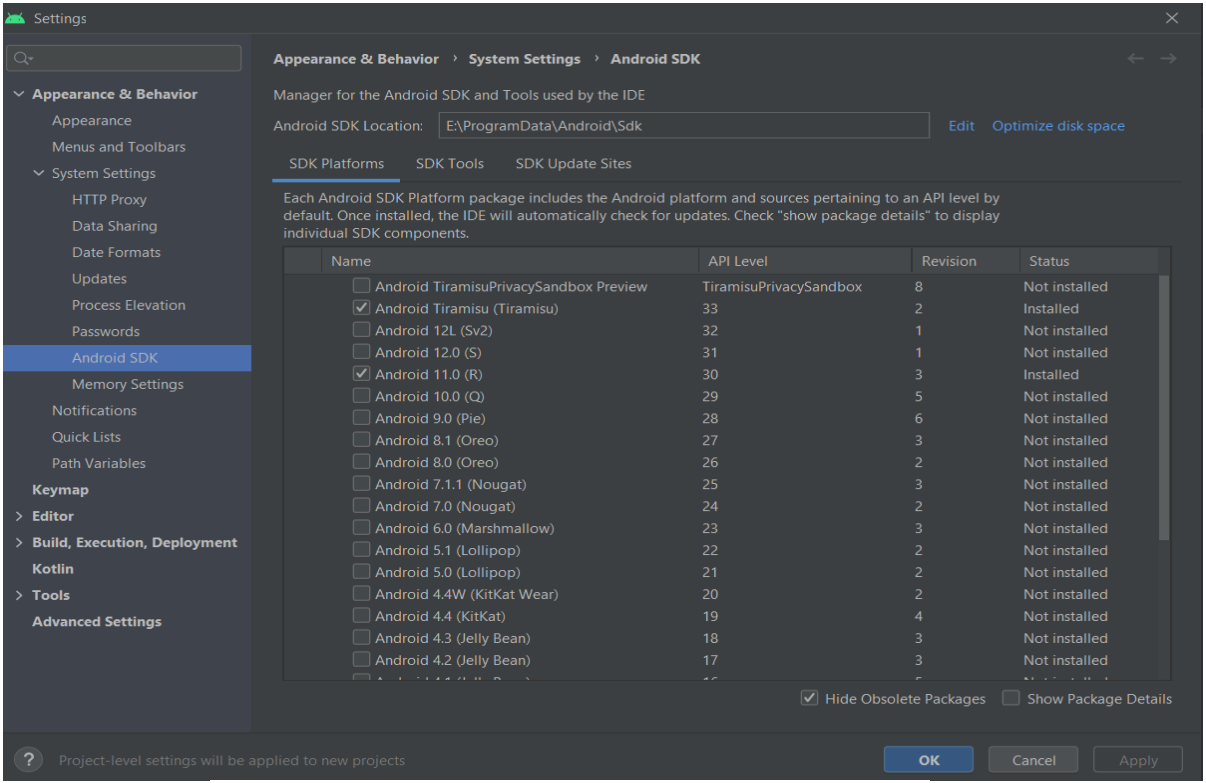


Рисунок 1.4 – Вибір версії ОС

Після встановлення SDK (це займе деякий час) слід створити віртуальний пристрій. Для цього заходимо до менеджеру пристроїв та створюємо новий віртуальний пристрій. (рис. 1.5)

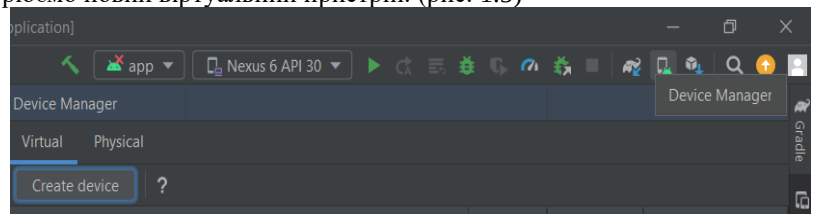


Рисунок 1.5 – Вибір Device Manager

Далі обираємо необхідний пристрій та завантажуємо певний його реліз. (рис. 1.6)

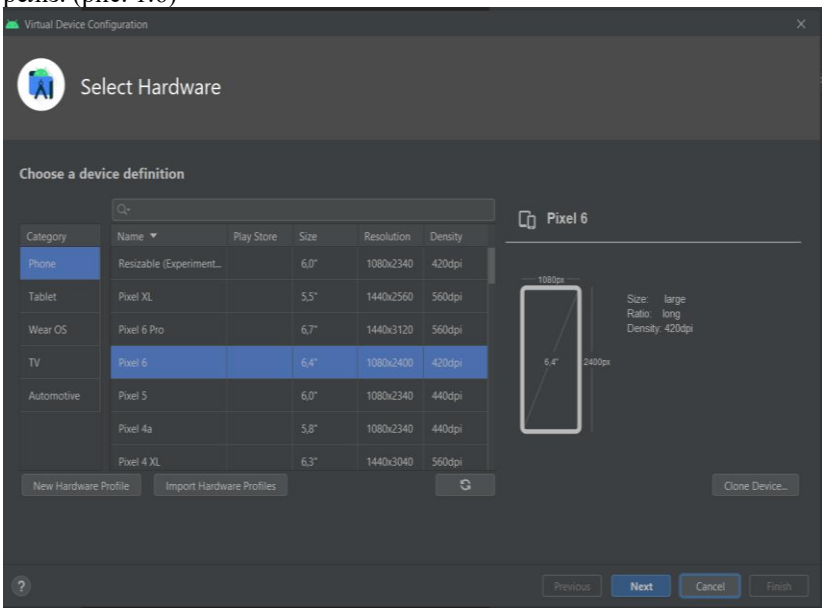


Рисунок 1.6 – Завантаження релізу

Після створення новий пристрій з’явиться у списку віртуальних пристроїв. (рис 1.7)

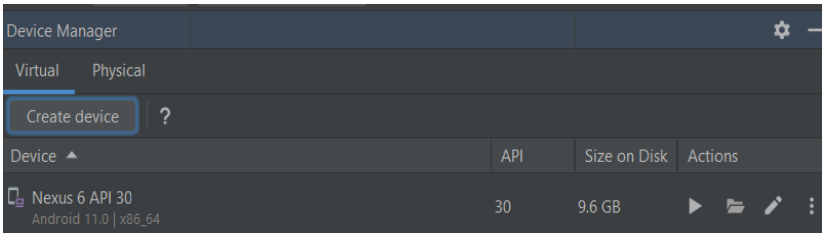


Рисунок 1.7 – Створений Android емулятор у списку пристроїв

Запустивши емулятор, ми побачимо віртуальний пристрій на екрані. (рис 1.8)

Рисунок 1.8 – Створений Android емулятор в робочому стані

Для того, щоб програмний застосунок одразу запускався на Android-емуляторі при запуску командою expo start у режимі a, слід створити змінну середовища оточення ANDROID_HOME та як значення прописати повний шлях до папки зі встановленим Android Sdk (**Мій комп'ютер** -> **Властивості** -> **Додаткові налаштування системи** -> **Змінні оточення**). (рис. 1.9)

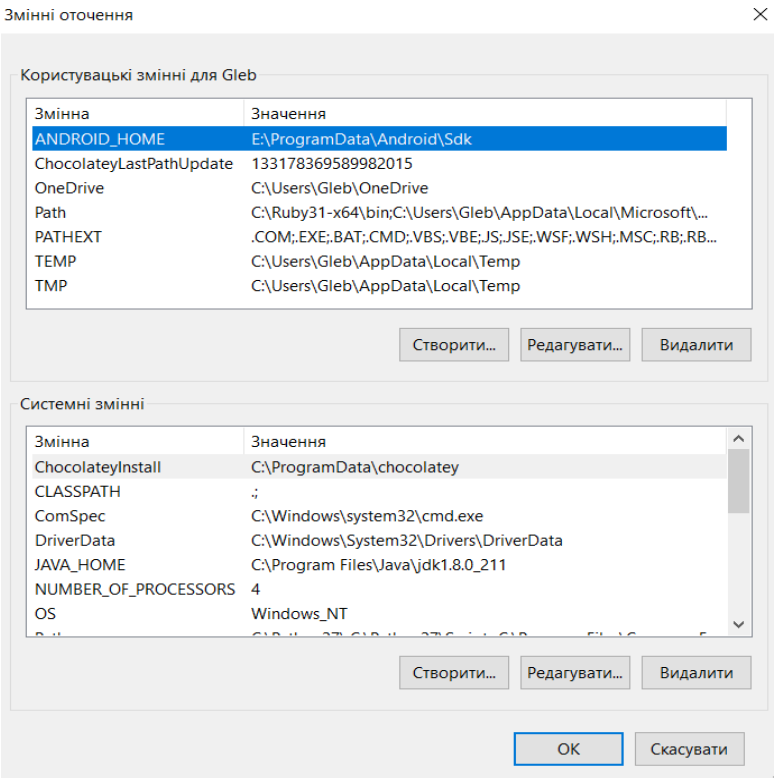


Рисунок 1.9 – Зміна змінних середовища оточення в Windows

Основний файл проєкту має назву App.js та має такий вміст:

```
import { StatusBar } from 'expo-status-bar';
import { StyleSheet, Text, View } from 'react-native';

export default function App() {
  return (
    <View style={styles.container}>
```

```
      <Text>Open up App.js to start working on your app!</Text>
      <StatusBar style="auto" />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

Запустимо застосунок на Android-емуляторі. (рис. 1.10)

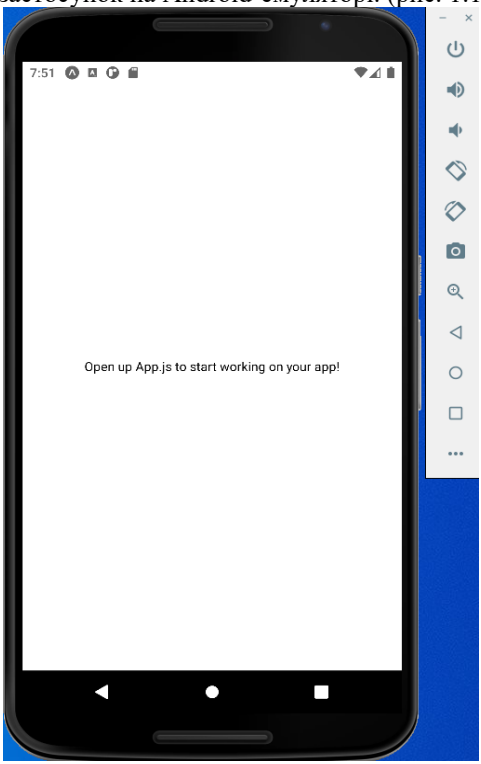


Рисунок 1.10 – Запуск застосунку на емуляторі Android

Завдання

Налаштувати середовище для подальшої роботи з React Native. Створити новий проєкт та запустити його на мобільному пристрої та емуляторі (Android або IOS).

Контрольні питання

1. Яка команда створює новий проєкт React Native?
2. На яких пристроях та яким чином можна запустити застосунок React Native?
3. Які налаштування потрібно зробити для можливості запуску на Android-симуляторі?

ЛАБОРАТОРНА РОБОТА №2 «ОСНОВНІ КОМПОНЕНТИ МОБІЛЬНОГО ЗАСТОСУНКУ REACT NATIVE (VIEW, TEXT, BUTTON, ALERT)»

Теоретичні відомості

View

Найбільш фундаментальний компонент для побудови інтерфейсу користувача, View – це контейнер, який підтримує компонування за допомогою flexbox, стилів, деяку обробку дотиків та елементи керування доступом. View відображає безпосередньо нативний еквівалент на будь-якій платформі, на якій працює React Native, чи то UIView, , android.view тощо.

Вигляд призначений для вкладеності в інші види і може мати від 0 до багатьох дочірніх елементів будь-якого типу.

У прикладі нижче створюється View, який обгортає два поля з кольором і текстовий компонент в ряд з відступами. (рис. 2.1)

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

export default function App() {
  return (
    <View
      style={{
        flexDirection: 'row',
        height: 100,
        padding: 20,
      }}
      >
      <View style={{backgroundColor: 'blue', flex: 0.3}} />
      <View style={{backgroundColor: 'red', flex: 0.5}} />
    </View>
  );
}
```

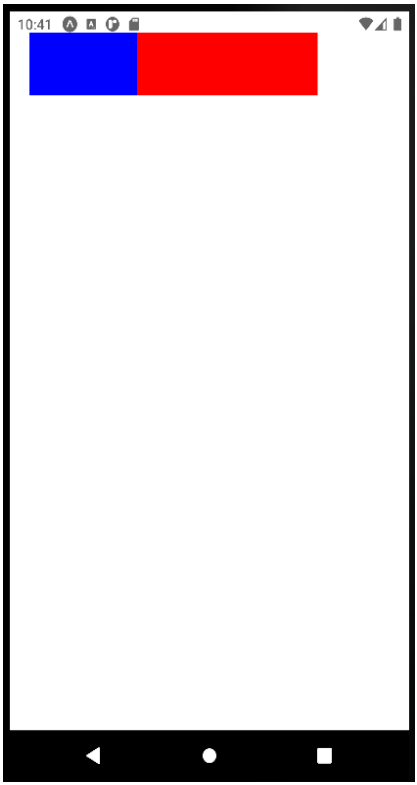


Рисунок 2.1 – Приклад створеного View

Text

React-компонент для відображення тексту. Text підтримує вкладеність, стилізацію та обробку дотиків. (рис. 2.2, рис. 2.3)

```
import React from "react";
import {StyleSheet, Text, SafeAreaView, StatusBar, Alert} from "react-native";

export default function App() {
  const handleTextPress = () => Alert.alert("Text is pressed");

  return (
    <SafeAreaView style={styles.container}>
```

```
            <Text numberOfLines={1} style={styles.text}
onPress={handleTextPress}>Hello world</Text>
            <StatusBar style="auto" />
        </SafeAreaView>
    )
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: '#fff',
    },
    text: {
        color: 'red'
    },
});
```

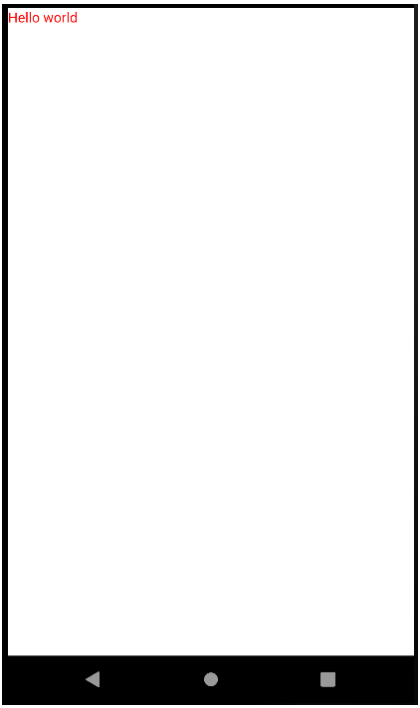


Рисунок 2.2 – Текстовий елемент на емуляторі

Рисунок 2.3 – Повідомлення виведене після натискання на текстовий фрагмент

Button

Більшість користувачів взаємодіють із мобільним застосунком за допомогою дотиків. Існують комбінації жестів, які спрацьовують на них, наприклад, тап по кнопці, масштабування карти, прокрутка списку тощо. Кнопка – це один із компонентів, який працює на натискання.

React Native Button – це базовий компонент, який працює при натисканні на нього. Він імпортує клас Button з react-native. (рис. 2.4).

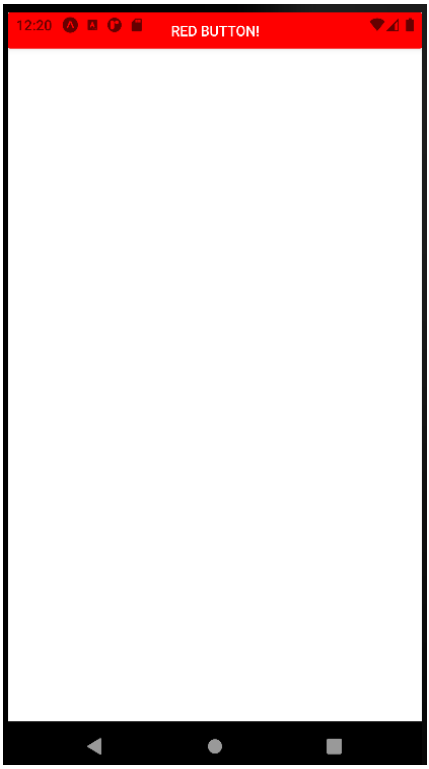


Рисунок 2.4 – Елемент Button

Властивість	**Тип**	**Обов'язкова**	**Опис**
onPress	function	так	Викликати обробник, коли користувач натискає кнопку.
title	string	так	Відображати текст всередині кнопки.
accessibilityLabel	string	ні	Відобразити текст для функцій доступності для незрячих.
color	Color	ні	Задати колір фону кнопки для Android або колір тексту для iOS.
disabled	bool	ні	Якщо встановлено, вимикає всі взаємодії для цього компонента.

Властивість	Тип	Обов'язкова	Опис
textID	string	ні	Використовується для визначення цього виду в наскрізних тестах.
hasTVPreferredFocus	bool	ні	Він надає перевагу роботі з фокусуванням на телевізорі лише для Apple TV.

```
import {Alert, Button} from "react-native";

export default function App() {
  const handleButtonPress = () => Alert.alert("Button is pressed");

  return (
    <Button
      onPress = {handleButtonPress}
      title = "Red button!"
      color = "red"
    />
  )
}
```

Touchable Opacity

Цей елемент змінює непрозорість елемента при дотику до нього. (рис. 2.5)

```
import React from "react";
import {Alert, TouchableOpacity, View, Text, StyleSheet} from "react-native";

export default function App() {
  const handleButtonPress = () => Alert.alert("Button is pressed");

  return (
    <View style = {styles.container}>
      <TouchableOpacity>
        <Text style = {styles.text} onPress={handleButtonPress()}>
          Button
        </Text>
      </TouchableOpacity>
    </View>
  )
}
```

```
const styles = StyleSheet.create ({
    container: {
        alignItems: 'center',
    },
    text: {
        borderWidth: 1,
        padding: 25,
        borderColor: 'black',
        backgroundColor: 'red'
    }
})
```

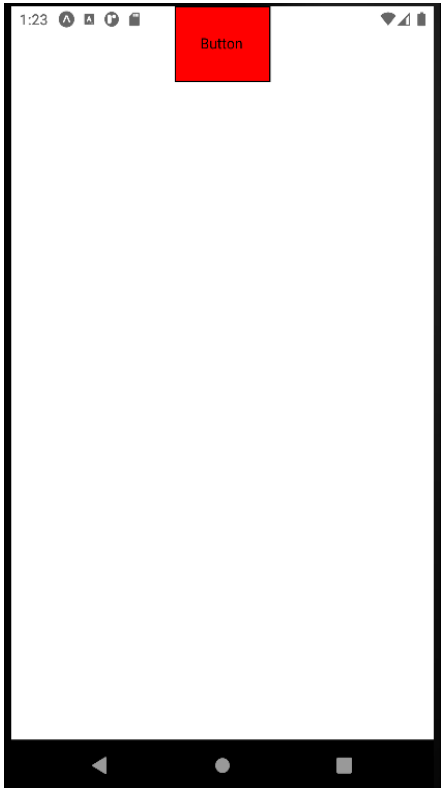


Рисунок 2.5 – Демонстрація Touchable Opacity

TouchableHighlight

TouchableHighlight є обгорткою для того, щоб елементи належним чином реагували на дотики. При натисканні зменшується непрозорість загорнутого представлення, що дозволяє просвічувати колір підкладки, затемнюючи або відтіняючи представлення.[3]

Підкладка з'являється внаслідок обгортання дочірнього подання у нове подання, що може вплинути на компонування, а іноді спричинити небажані візуальні артефакти, якщо її неправильно використовувати, наприклад, якщо backgroundColor обгорнутого подання явно не встановлено на непрозорий колір.

TouchableHighlight повинен мати один дочірній компонент (не нуль і не більше одного).

Попередній приклад можна переписати з використанням TouchableHighlight:

```
……
return (
    <TouchableHighlight activeOpacity={0.6}
underlayColor="#DDDDDD"  onPress={handleButtonPress>
    <Text style = {styles.text}>
        Button
     </Text>
      </TouchableHighlight >
  )
```

Alert

React Native Alert – це API, який використовується для відображення діалогового вікна попередження із заданим заголовком та повідомленням. Він використовує метод alert() для виклику діалогового вікна попередження. Це діалогове вікно містить три різні списки кнопок (комбінації нейтральних, негативних і позитивних) для виконання дій. Натискання будь-якої із цих кнопок викликає метод зворотного виклику onPress і закриває сповіщення. За замовчуванням, Сповіщення має лише позитивну кнопку (ОК).

В iOS ми можемо вказати кількість кнопок. Кожній кнопці можна окремо задати стиль – за замовчуванням, скасування або деструктивний.

В Android кнопка «Сповіщення» може бути задана трьома кнопками. Це нейтральна, негативна та позитивна кнопки.

Кнопки сповіщення можна вказати у трьох різних комбінаціях. Це такі:

– Якщо вказати лише одну кнопку, вона буде «позитивною» (наприклад, «ОК»).

– Якщо ми вкажемо дві кнопки, вони будуть «негативними» і «позитивними» (наприклад, «Скасувати», «ОК»).

– Вказівка трьох кнопок означає «нейтральний», «негативний», «позитивний» (наприклад, «Пізніше», «Скасувати», «ОК»).

В Android сповіщення за замовчуванням можна відхилити, натиснувши за межами діалогового вікна сповіщення. Подію відхилення можна обробити за допомогою необов'язкового параметра з властивістю зворотного виклику onDismiss { onDismiss: () => {} }. Втім, ми можемо вимкнути властивість відхилення за допомогою властивості {cancelable: false}.

Приклад 1. (рис. 2.6)

```
import React from "react";
import {Alert, Button, View, StyleSheet} from "react-native";

export default function App() {
   const handleButtonPress = () => Alert.alert(
      'Alert Title',
      'My Alert Msg',
      [
         {
            text: 'Cancel',
            onPress: () => console.log('Cancel Pressed'),
            style: 'cancel',
         },
         {text: 'OK', onPress: () => console.log('OK Pressed')},
      ]
   );
   return (
      <View style={styles.container}>
         <View style={styles.buttonContainer}>
            <Button
               onPress={handleButtonPress}
               title="Button"
            />
         </View>
      </View>
   )
}

const styles = StyleSheet.create ({
```

```
    container: {
      flex: 1,
      justifyContent: 'center',
    },
    buttonContainer: {
      margin: 20
    }
})
```

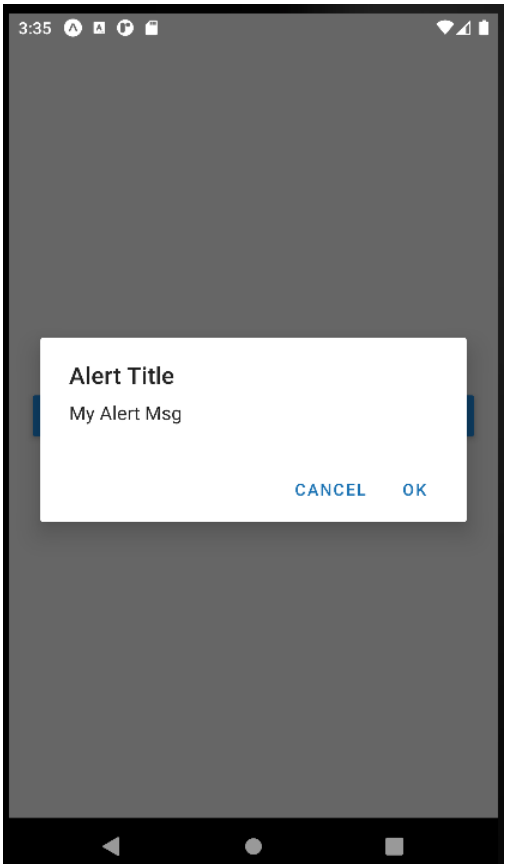


Рисунок 2.6 – Приклад 1 використання Alert

Для того, щоб побачити повідомлення в консолі браузера Google Chrome, слід натиснути F12. (рис. 2.7)

Рисунок 2.7 – Сповіщення в консолі Google Chrome

Приклад 2. (рис. 2.8)

```
import React from "react";
import {Alert, Button, View, StyleSheet} from "react-native";

export default function App() {
   const handleButtonPress = () =>  Alert.alert(
         'Alert Title',
         'My Alert Msg',
         [
            {text: 'Ask me later', onPress: () => console.log('Ask me later
pressed')},
            {
               text: 'Cancel',
               onPress: () => console.log('Cancel Pressed'),
               style: 'cancel',
            },
            {text: 'OK', onPress: () => console.log('OK Pressed')},
         ],
         {cancelable: false}
   );
   return (
      <View style={styles.container}>
         <View style={styles.buttonContainer}>
            <Button
               onPress={handleButtonPress}
               title="Button"
            />
         </View>
      </View>
   )
}

const styles = StyleSheet.create ({
   container: {
      flex: 1,
      justifyContent: 'center',
```

```
    },
    buttonContainer: {
      margin: 20
    }
})
```

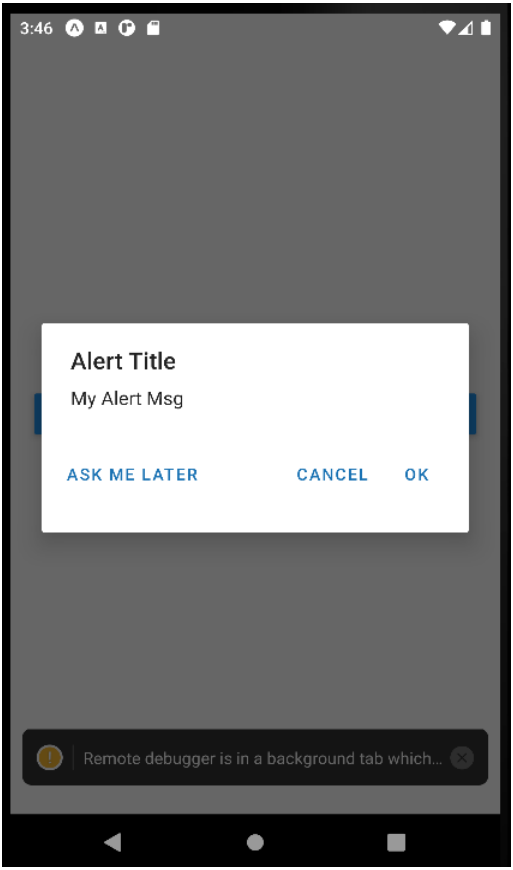


Рисунок 2.8 – Демонстрація сповіщення за прикладом 2

Завдання

1. Створіть текстовий напис і при натисканні на нього викликайте просте спливаюче вікно.
2. Додайте до текстового напису додатковий об’єкт, який створюватиме ефект натискання.

3. Створіть у застосунку спливаюче вікно при натисканні на текст або кнопку.

Контрольні питання

1. Які можливості має компонент View? Як він представляється у застосунку залежно від платформи?
2. Як додати до мобільного застосунку статичний текст? Чи можна обробити натискання на текст?
3. Як додати до мобільного застосунку кнопку? Які основні властивості має компонент кнопки?
4. Для чого призначено компонент Alert? Які різновиди він має?

ЛАБОРАТОРНА РОБОТА №3 «КЕРУВАННЯ ДАНИМИ КОМПОНЕНТІВ У ЗАСТОСУНКАХ REACT NATIVE»

Теоретичні відомості

Дані всередині React-компонентів управляються за допомогою стану та реквізитів.

Різниця між станом і реквізитом

Стан є змінним, тоді як реквізити є незмінними. Це означає, що стан може бути оновлений у майбутньому, в той час як реквізити не можуть бути оновлені.

Використання стану

Це наш кореневий компонент. Ми просто імпортуємо Home, який буде використовуватися в більшості розділів.

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default class App extends React.Component {
  state = {
    myState: '\n\nLorem ipsum dolor sit amet, consectetur adipisicing elit, used do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu  fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.'
  }

  render() {
    return (
    <View>
      <Text> {this.state.myState} </Text>
    </View>
    );
  }
}
```

Ми бачимо в емуляторі текст зі стану, як на наступному скріншоті. (рис. 3.1)

Рисунок 3.1 – Демонстрація стану

Оновлення стану

Оскільки стан є змінним, ми можемо оновити його, створивши функцію updateState і викликавши її за допомогою події onPress = {updateState}.

```
import React, { Component } from 'react'
import { Text, View } from 'react-native'

class Home extends Component {
  state = {
    myState: '\n\nLorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.'
  }

  updateState = () => this.setState({ myState: '\n\nThe state is updated' })

  render() {
    return (
      <View>
        <Text onPress = {this.updateState}>
          {this.state.myState}
        </Text>
      </View>
    );
  }
}

export default Home;
```

У цьому прикладі замість компоненту за замовчуванням використовується компонент Home. Для того, щоб включити його у проєкт, необхідно створити новий файл Home.js, а потім у файлі index.js задати його у якості компоненту за замовчуванням.(рис. 3.2, рис. 3.3)

```
import {AppRegistry} from 'react-native';
import Home from './Home';
import {name as appName} from './app.json';
```

```
AppRegistry.registerComponent(appName, () => Home);
```

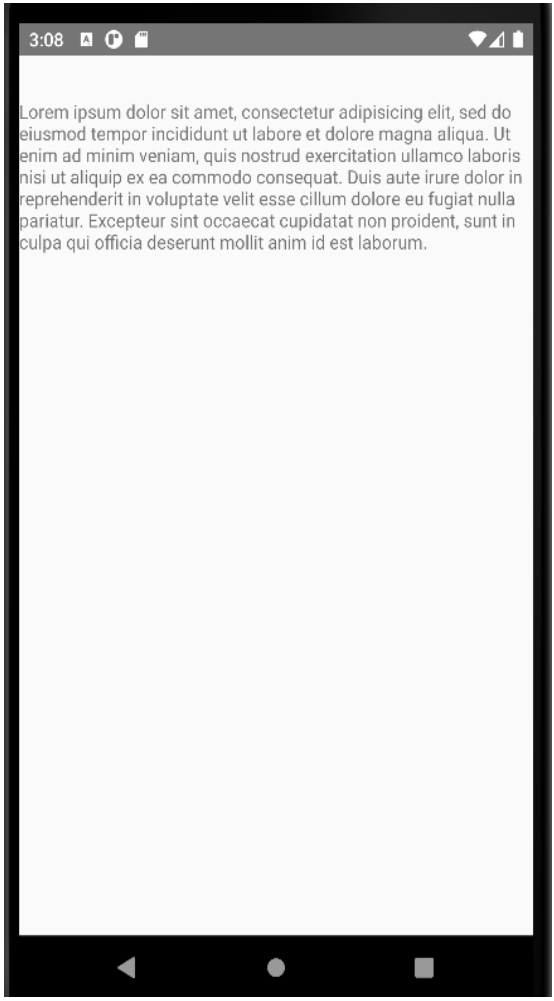


Рисунок 3.2 – До натиснення на текст

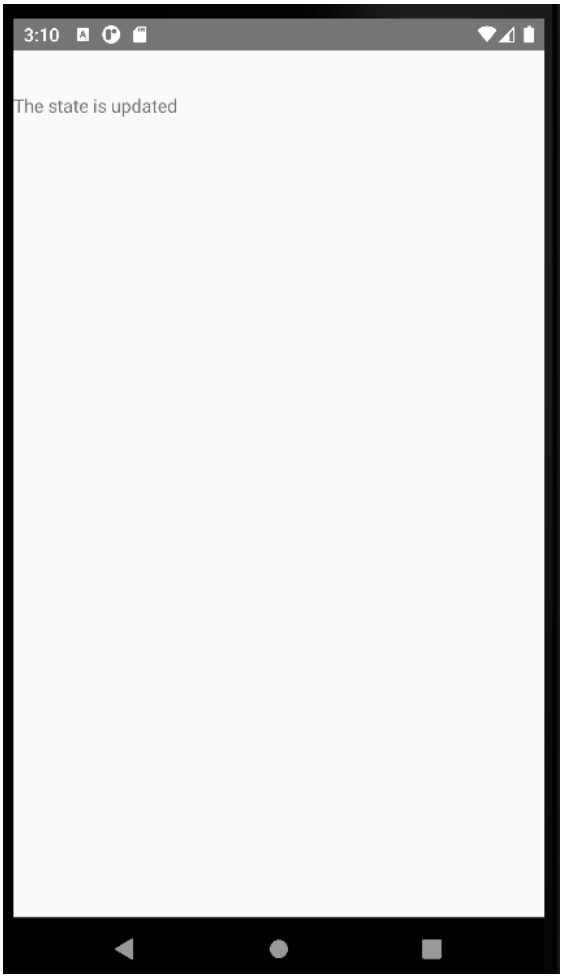


Рисунок 3.3 – Після натиснення на текст

Також розглянемо, як використовувати синтаксис стрілочної функції для updateState. Слід пам’ятати, що цей синтаксис використовує лексичну область видимості, і це ключове слово буде прив’язано до об’єкта оточення (Class). Іноді це може призвести до неочікуваної поведінки.[4]

Іншим способом визначення методів є використання функцій ЕС5, але в цьому випадку нам потрібно буде прив'язати їх вручну в конструкторі. Розглянемо наступний приклад, щоб зрозуміти це.

```
import React, { Component } from 'react'
import { Text, View } from 'react-native'

class Home extends Component {
  constructor() {
    super()
    this.updateState = this.updateState.bind(this)
    this.state = {
      myState: '\n\nLorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.'
    }
  }

  updateState()  {
    this.setState({ myState: '\n\nThe state is updated' })
  }

  render() {
    return (
      <View>
        <Text onPress = {this.updateState}>
          {this.state.myState}
        </Text>
      </View>
    );
  }
}

export default Home;
```

Презентаційні компоненти повинні отримувати всі дані через передачу реквізитів (props). Тільки контейнерні компоненти повинні мати стан.

Контейнерний компонент

Тепер ми зрозуміємо, що таке контейнерний компонент і як він працює.

Далі ми оновимо наш контейнерний компонент. Цей компонент буде обробляти стан і передавати реквізити до компонента презентації.

Контейнерний компонент використовується тільки для обробки стану. Вся функціональність, пов'язана з виглядом (стилізацією і т.д.), буде оброблятися в презентаційному компоненті.

Якщо ми хочемо використати попередній приклад, нам потрібно видалити елемент Text з функції рендерінгу, оскільки цей елемент використовується для представлення тексту користувачам. Він повинен знаходитися всередині презентаційного компонента.

Давайте розглянемо код у прикладі, наведеному нижче. Ми імпортуємо компонент PresentationalComponent і передаємо його функції render.

Після того, як ми імпортували PresentationalComponent і передали його функції render, нам потрібно передати реквізити. Ми передамо реквізити, додавши myText = {this.state.myText} і deleteText = {this.deleteText} до <PresentationalComponent>. Тепер ми зможемо отримати доступ до них всередині презентаційного компонента.

```
import React, { Component } from 'react'
import { Text, View } from 'react-native'
import PresentationalComponent from "./PresentationalComponent";

class Home extends Component {
  state = {
    myState: '\n\nLorem ipsum dolor sit amet, consectetur adipisicing elit, used do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.'
  }
  updateState = () => this.setState({ myState: "\n\nThe state is updated" })

  render() {
    return (
      <View>
```

```
        <PresentationalComponent myState = {this.state.myState}
updateState = {this.updateState}/>
      </View>
    );
  }

}

export default Home;
```

Презентаційний компонент

Зараз ми з'ясуємо, що таке презентаційний компонент і як він працює.

Презентаційні компоненти слід використовувати лише для представлення користувачам. Ці компоненти не мають стану. Вони отримують всі дані та функції як реквізити.

Найкращою практикою є використання якомога більшої кількості презентаційних компонентів.

Наш компонент отримуватиме реквізити, повертатиме елементи представлення, представлятиме текст за допомогою {props.myState} і викликатиме функцію {props.updateState}, коли користувач клацне на тексті.

```
import React, { Component } from 'react'
import { Text, View } from 'react-native'

const PresentationalComponent = (props) => {
  return (
    <View>
      <Text onPress = {props.updateState}>
        {props.myState}
      </Text>
    </View>
  )
}
export default PresentationalComponent
```

Тепер ми маємо той самий функціонал, що і у прикладі з використання стану. Єдина відмінність полягає в тому, що ми зробили рефакторинг коду до контейнера і презентаційного компонента.

Ви можете запустити додаток і побачити текст. При натисненні текст буде видалений з екрану.

Завдання

1. Створити компонент Button, який виводитиме лічильник кількості її натискань у статичний текст.

Контрольні питання

1. Які способи керування даними існують у React Native? У чому їх різниця?
2. Охарактеризуйте поняття стану. До чого він може застосовуватись?
3. Охарактеризуйте поняття реквізитів. До чого вони можуть застосовуватись?
4. Для чого доцільне застосування контейнерного компоненту?
5. Для чого доцільне застосування презентаційного компоненту?

ЛАБОРАТОРНА РОБОТА №4
«ВИКОРИСТАННЯ КОМПОНЕНТІВ SCROLLVIEW ТА TEXTINPUT У ЗАСТОСУНКУ REACT NATIVE»

ScrollView

ScrollView – це загальний контейнер, який прокручує декілька дочірніх компонентів та подань всередині нього. У ScrollView ми можемо прокручувати компоненти в обох напрямках – вертикальному та горизонтальному. За замовчуванням, контейнер ScrollView прокручує свої компоненти та подання у вертикальному напрямку. Для прокрутки компонентів по горизонталі він використовує пропси horizontal: true (за замовчуванням horizontal: false).[5]

У цьому прикладі ми створюємо вертикальне ScrollView з використанням компонентів Text і Button. (рис. 4.1)

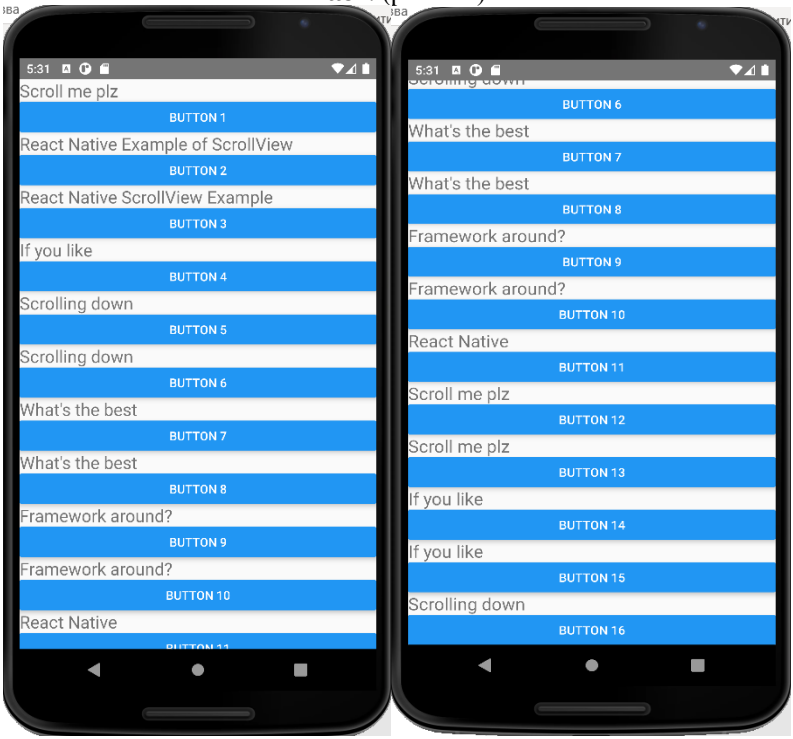


Рисунок 4.1 – Використання ScrollView в застосунку

```
import React, {Component} from 'react';
import {ScrollView, Text, Button} from 'react-native';

export default class ScrolledView extends Component {
  onPressButton() {
    alert('You clicked the button!')
  }

  render() {
    return(
      <ScrollView>
        <Text style={{fontSize: 20}}>Scroll me plz</Text>
        <Button title={'Button 1'} onPress={this.onPressButton}/>
        <Text style={{fontSize: 20}}>React Native Example of ScrollView</Text>
        <Button title={'Button 2'} onPress={this.onPressButton}/>
        <Text style={{fontSize: 20}}>React Native ScrollView Example</Text>
        <Button title={'Button 3'} onPress={this.onPressButton}/>
        <Text style={{fontSize: 20}}>If you like</Text>
        <Button title={'Button 4'} onPress={this.onPressButton}/>
        <Text style={{fontSize: 20}}>Scrolling down</Text>
        <Button title={'Button 5'} onPress={this.onPressButton}/>
        <Text style={{fontSize: 20}}>Scrolling down</Text>
        <Button title={'Button 6'} onPress={this.onPressButton}/>
        <Text style={{fontSize: 20}}>What's the best</Text>
        <Button title={'Button 7'} onPress={this.onPressButton}/>
        <Text style={{fontSize: 20}}>What's the best</Text>
        <Button title={'Button 8'} onPress={this.onPressButton}/>
        <Text style={{fontSize: 20}}>Framework around?</Text>
        <Button title={'Button 9'} onPress={this.onPressButton}/>
        <Text style={{fontSize: 20}}>Framework around?</Text>
        <Button title={'Button 10'} onPress={this.onPressButton}/>
        <Text style={{fontSize: 20}}>React Native</Text>
        <Button title={'Button 11'} onPress={this.onPressButton}/>
        <Text style={{fontSize: 20}}>Scroll me plz</Text>
        <Button title={'Button 12'} onPress={this.onPressButton} />
        <Text style={{fontSize: 20}}>Scroll me plz</Text>
        <Button title={'Button 13'} onPress={this.onPressButton}/>
        <Text style={{fontSize: 20}}>If you like</Text>
        <Button title={'Button 14'} onPress={this.onPressButton}/>
```

```
        <Text style={{fontSize: 20}}>If you like</Text>
        <Button title={'Button 15'} onPress={this.onPressButton}/>
        <Text style={{fontSize: 20}}>Scrolling down</Text>
        <Button title={'Button 16'} onPress={this.onPressButton}/>
      </ScrollView>
    );
  }
}
```

Горизонтальний ScrollView

Горизонтальний ScrollView прокручує компоненти та представлення зліва направо. Його буде реалізовано, якщо встановити реквізит horizontal у значення true (horizontal={true}). (рис. 4.2)

```
import React, {Component} from 'react';
import {StyleSheet, ScrollView, View, Text, Button} from 'react-native';

export default class ScrolledView extends Component {
  onPressButton() {
    alert('You clicked the button!')
  }

  render() {
    return(
      <ScrollView horizontal={true} style={styles.container}>
        <Text style={{fontSize:22, padding: 10}}>Horizontal ScrollView</Text>
        <View style={[{ width: 220,height: 70,padding: 10 }]}>
          <Button
            onPress={this.onPressButton}
            title="Button 1"
            color="#FF3D00"
          />
        </View>
        <Text style={{fontSize:22, padding: 10}}>next button</Text>
        <View style={[{ width: 220,height: 70,padding: 10 }]}>
          <Button
            onPress={this.onPressButton}
            title="Button 2"
            color="#3D00FF"
          />
```

```
          </View>
          <Text style={{fontSize:22, padding: 10}}>React Native
ScrollView Example</Text>
          <View style={[{ width: 220,height: 70,padding: 10 }]}>
            <Button
              onPress={this.onPressButton}
              title="Button 3"
              color="#FFFF3D"
            />
          </View>
          <Text style={{fontSize:22, padding: 10}}>If you like</Text>
          <View style={[{ width: 220,height: 70,padding: 10 }]}>
            <Button
              onPress={this.onPressButton}
              title="Button 4"
              color="#FF3DFF"
            />
          </View>
          <Text style={{fontSize:22, padding: 10}}>Scrolling
horizontal</Text>
          <View style={[{ width: 220,height: 70,padding: 10 }]}>
            <Button
              onPress={this.onPressButton}
              title="Button 5"
              color="#3DFF00"
            />
          </View>
        </ScrollView>
      );
    }
  }

  const styles = StyleSheet.create({
    container:{
      flex: 1,
    },
  })
```

Рисунок 4.2 – Використання горизонтального ScrollView в застосунку

Text Input

TextInput – це основний компонент для введення тексту. Він має кілька реквізитів, які налаштовують різні функції, наприклад, onChangeText, який отримує функцію і викликає її щоразу, коли текст змінюється. Проп onSubmitEditing отримує функцію, яка викликається при надсиланні тексту.

У наступному прикладі спочатку ми визначимо початковий стан, після чого створимо функції handleEmail та handlePassword. Ці функції використовуються для оновлення стану.

Функція login() просто сповіщатиме про поточне значення стану.

Ми також додамо деякі інші властивості до текстового вводу, щоб вимкнути автоматичне введення великих літер, видалити нижню межу на пристроях Android та встановити заповнювач. (рис. 4.3)

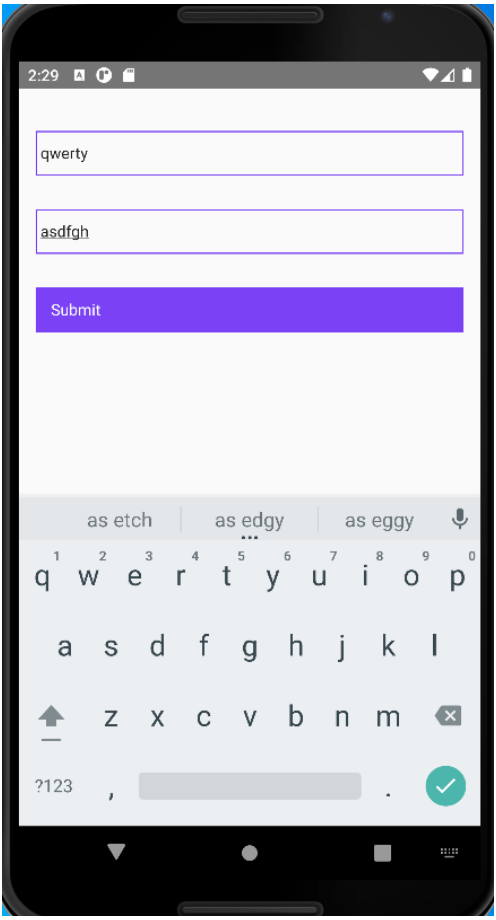


Рисунок 4.3 – TextInput компоненти

```
import React, { Component } from 'react'
import { View, Text, TouchableOpacity, TextInput, StyleSheet } from 'react-native'

class App extends Component {
  state = {
    email: '',
    password: ''
  }
```

```
  handleEmail = (text) => {
    this.setState({ email: text })
  }
  handlePassword = (text) => {
    this.setState({ password: text })
  }
  login = (email, pass) => {
    alert('email: ' + email + ' password: ' + pass)
  }
  render() {
    return (
      <View style = {styles.container}>
        <TextInput style = {styles.input}
          underlineColorAndroid = "transparent"
          placeholder = "Email"
          placeholderTextColor = "#9a73ef"
          autoCapitalize = "none"
          onChangeText = {this.handleEmail}/>

        <TextInput style = {styles.input}
          underlineColorAndroid = "transparent"
          placeholder = "Password"
          placeholderTextColor = "#9a73ef"
          autoCapitalize = "none"
          onChangeText = {this.handlePassword}/>

        <TouchableOpacity
          style = {styles.submitButton}
          onPress = {
            () => this.login(this.state.email, this.state.password)
          }>
          <Text style = {styles.submitButtonText}> Submit </Text>
        </TouchableOpacity>
      </View>
    )
  }
}
export default App

const styles = StyleSheet.create({
  container: {
```

```
    paddingTop: 23
  },
  input: {
    margin: 15,
    height: 40,
    borderColor: '#7a42f4',
    borderWidth: 1
  },
  submitButton: {
    backgroundColor: '#7a42f4',
    padding: 10,
    margin: 15,
    height: 40,
  },
  submitButtonText:{
    color: 'white'
  }
})
```

Завдання

1. Створити ScrollView з візуальних елементів, вивчених раніше, на ваш розсуд.
2. Динамічно вивести текст із компонента TextInput у компонент Text.

Контрольні питання

1. Для чого призначений візуальний елемент ScrollView?
2. Як задається напрям прокрутки ScrollView?
3. Для чого призначені реквізити onChangeText та onSubmitEditing візуального елементу TextInput?
4. Перелічіть основні реквізити TextInput, що призначені для налаштування.

ЛАБОРАТОРНА РОБОТА №5
«СТВОРЕННЯ МОБІЛЬНОГО ЗАСТОСУНКА-ОРГАНАЙЗЕРА ЗА ДОПОМОГОЮ ЕЛЕМЕНТА LISTVIEW»

Теоретичні відомості
React Native ListView

React Native ListView – це компонент представлення, який містить список елементів і відображається у вигляді вертикального списку з прокруткою. Мінімальним API для створення списку є ListView.DataSource. Він заповнює простий масив блоків даних і створює компонент ListView із джерелом даних і функцією зворотного виклику renderRow. Функція renderRow отримує блок із масиву даних і повертає компонент, який можна відрендерити.

Примітка: Компонент ListView є застарілим. Для реалізації компонента списку використовуйте нові компоненти списків FlatList або SectionList.

Приклад React Native ListView 1

Давайте створимо приклад компонента ListView. У цьому прикладі ми створюємо джерело даних і заповнюємо його масивом блоків даних. Створіть компонент ListView, використовуючи цей масив як джерело даних, і передайте його у функції зворотного виклику renderRow. renderRow – це функція, яка повертає компонент, що рендерить рядок. (рис. 5.1)

```
import React, { Component } from 'react'
import { Text, ListView, StyleSheet } from 'react-native'

export default class MyListComponent extends Component {
  constructor() {
    super();
    const ds = new ListView.DataSource({rowHasChanged: (r1, r2) =>
r1 !== r2});
    this.state = {
      dataSource: ds.cloneWithRows(['Android','iOS', 'Java','Php',
'Hadoop',
        'Sap', 'Python','Ajax', 'C++',
        'Ruby', 'Rails','.Net', 'Perl']),
    };
  }
```

```
    render() {
        return (
            <ListView
                dataSource={this.state.dataSource}
                renderRow={
                    (rowData) =>
                        <Text style={{fontSize: 20}}>{rowData}</Text>}
            />
        );
    }
}
```

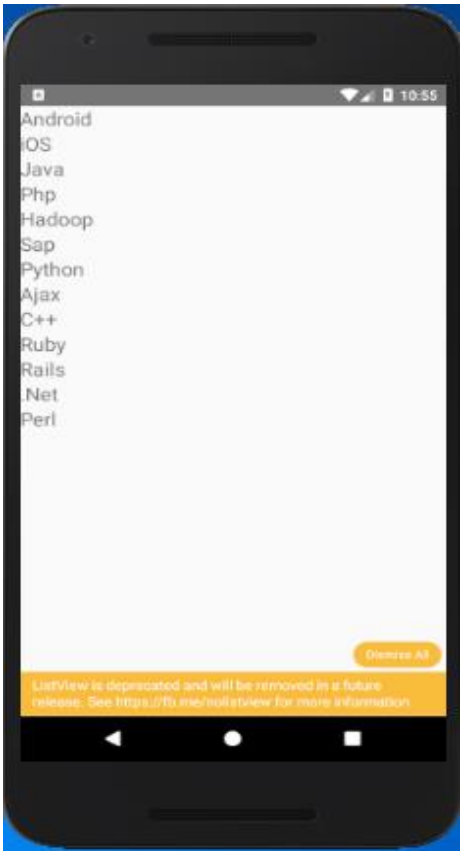


Рисунок 5.1 – Приклад списку 1

У вищенаведеному коді ми створюємо екземпляр ListViewDataSource у конструкторі. ListViewDataSource є параметром і приймає аргумент, який містить будь-який з цих чотирьох:

getRowData(dataBlob, sectionID, rowID)
getSectionHeaderData(dataBlob, sectionID)
rowHasChanged(previousRowData, nextRowData)
sectionHeaderHasChanged(previousSectionData, nextSectionData)

Додавання розділення та виконання дій над елементами ListView

Розділення додається для створення окремого блоку і кращого відображення елементів списку. Проп renderSeparator використовується для додавання роздільника між елементами (даними) ListView.[6]

Реалізуємо пропси onPress над Text для виконання дії при натисканні на елементи списку. Тут ми генеруємо попереджувальне повідомлення і відображаємо елементи списку, на яких користувач натискає.

```
import React from 'react';
import { View, ListView, StyleSheet, Text,Alert} from 'react-native';

class ListViewDemo extends React.Component {
  constructor(props) {
    super(props);

    const ds = new ListView.DataSource({rowHasChanged: (r1, r2) =>
r1 !== r2});
    this.state = {
      dataSource: ds.cloneWithRows([ "Android",
        "iOS","Java","Swift",
        "Php","Hadoop","Sap",
        "Python","Ajax", "C++",
        "Ruby", "Rails",".Net",
        "Perl",
      ])
    };
  }
  //handling onPress action
  getListViewItem = (rowData) => {
    Alert.alert(rowData);
  }
  render() {
    return (
        <ListView
```

```
                style={styles.container}
                dataSource={this.state.dataSource}
                renderRow={(rowData) =>
                  <Text style={styles.rowViewContainer}
                      onPress={this.getListViewItem.bind(this,
rowData)}>{rowData}
                  </Text>
                }
                renderSeparator={(sectionId, rowId) =>
                  <View key={rowId} style={styles.separator} />}//adding
separation
            />
        );
    }
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: "#e5e5e5"
    },
    separator: {
        height: 0.5, width: "100%", backgroundColor: "#000"
    },
    rowViewContainer: {
        flex: 1,
        paddingRight: 15,
        paddingTop: 13,
        paddingBottom: 13,
        borderBottomWidth: 0.5,
        borderColor: '#c9c9c9',
        flexDirection: 'row',
        alignItems: 'center',
        fontSize: 20,
        marginLeft: 10,
    },
});

export default ListViewDemo;
```

React Native FlatList

Компонент FlatList відображає подібні структуровані дані у вигляді списку з можливістю прокрутки. Він добре працює для великих списків даних, де кількість елементів списку може змінюватися з часом. FlatList показує тільки ті елементи рендерингу, які в даний момент відображаються на екрані, а не всі елементи списку одразу.

Компонент FlatList отримує два обов'язкових пропси: data і renderItem.

Дані є джерелом елементів для списку, а renderItem бере один елемент із джерела і повертає відформатований компонент для рендерингу.

Для реалізації компонента FlatList нам потрібно імпортувати FlatList з бібліотеки 'react-native'.

Приклад React Native FlatList

У цьому прикладі ми надаємо жорстко закодовані елементи в проп даних. Кожен елемент у пропсах даних рендериться як текстовий компонент.

Проп ItemSeparatorComponent FlatList використовується для реалізації роздільника між елементами списку. Для виконання події кліку на елементах списку ми використовуємо проп onPress до Text. (рис. 5.2)

```
import React, { Component } from 'react';
import { AppRegistry, FlatList,
   StyleSheet, Text, View,Alert } from 'react-native';

export default class FlatListBasics extends Component {

   renderSeparator = () => {
      return (
         <View
            style={{
               height: 1,
               width: "100%",
               backgroundColor: "#000",
            }}
         />
      );
   };
   //handling onPress action
   getListViewItem = (item) => {
```

```
        Alert.alert(item.key);
    }

    render() {
        return (
            <View style={styles.container}>
                <FlatList
                    data={[
                        {key: 'Android'},{key: 'iOS'}, {key: 'Java'},{key:
'Swift'},
                        {key: 'Php'},{key: 'Hadoop'},{key: 'Sap'},
                        {key: 'Python'},{key: 'Ajax'}, {key: 'C++'},
                        {key: 'Ruby'},{key: 'Rails'},{key: '.Net'},
                        {key: 'Perl'},{key: 'Sap'},{key: 'Python'},
                        {key: 'Ajax'}, {key: 'C++'},{key: 'Ruby'},
                        {key: 'Rails'},{key: '.Net'},{key: 'Perl'}
                    ]}
                    renderItem={({item}) =>
                        <Text style={styles.item}
                            onPress={this.getListViewItem.bind(this,
item)}>{item.key}</Text>}
                    ItemSeparatorComponent={this.renderSeparator}
                />
            </View>
        );
    }
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
    },
    item: {
        padding: 10,
        fontSize: 18,
        height: 44,
    },
})
```

AppRegistry.registerComponent('AwesomeProject', () => FlatListBasics);

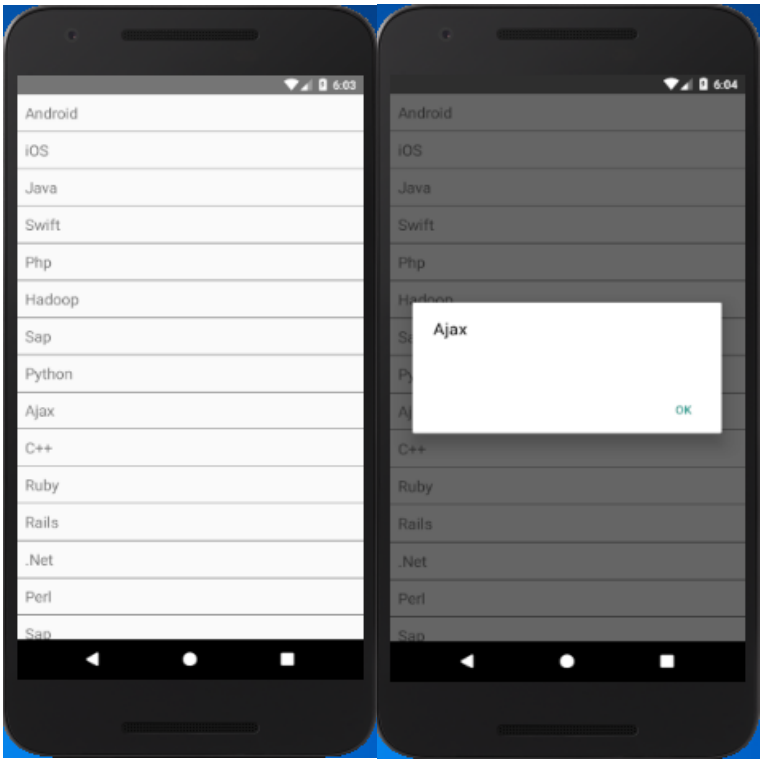


Рисунок 5.2 – Приклад FlatList

Завдання

Створити мобільний застосунок органайзеру, що представляє собою список справ на поточний день. Застосунок повинен мати можливість додавання нової справи та видалення вже виконаних справ.

Контрольні запитання

1. Які елементи керування списками у React Native ви знаєте?
2. Яким способом можна відобразити дані у елементі ListView?
3. Яким способом можна відобразити дані у елементі FlatList?
4. Для чого використовується реквізит propSeparator?

ЛАБОРАТОРНА РОБОТА №6
«СТИЛІЗАЦІЯ ЕЛЕМЕНТІВ У REACT NATIVE»

Таблиця стилів (stylesheet) у React Native – це абстракція, аналогічна таблицям стилів CSS. Стилізація в React Native дуже схожа на стилізацію в Web з використанням CSS з невеликими відмінностями.

Якщо говорити коротко, то верстка з Flexbox дає нам прості рішення колись непростих завдань. Наприклад, коли потрібно вирівняти елемент по вертикалі, або притиснути підвал до низу екрана, або просто вставити кілька блоків в один ряд, так щоб вони займали весь вільний простір. Подібні завдання вирішуються і без flex. Але, як правило, ці рішення більше схожі на «милиці» – прийоми використання css не за призначенням. Тоді як із flexbox такі завдання вирішуються саме так, як задумує flex-модель.

Стилі в React Native дають змогу поліпшити UI/UX вашого застосунку.

StyleSheet.create() дозволяє створити стилі для ваших компонентів.

Умовний оператор дозволяє задавати стилі вашим компонентам залежно від параметрів.

При необхідності додавання декількох стилів вашому компоненту – стилі обертаються в масив.

```
import React, {useState} from 'react';
import {Text, View, SafeAreaView, Button} from 'react-native';
import styles from './styles';

const App = () => {
  const [active, setActive] = useState(false);
  return (
    <SafeAreaView style={styles.safeAreaContainer}>
      <View style={styles.mainContainer}>
        <View style={styles.firstContainer}>
          <Text>first</Text>
        </View>
        <View style={styles.secondContainer}>
          <Text>second</Text>
        </View>
        <View style={active ? styles.thirdContainer : styles.secondContainer}>
          <Text>third</Text>
```

```
            </View>
            <Button title="press me" onPress={() => setActive(!active)}
/>
          </View>
        </SafeAreaView>
      );
    };

    export default App;


    import {StyleSheet} from 'react-native';

    const styles = StyleSheet.create({
      safeAreaContainer: {
        flex: 1,
        display: 'flex',
        backgroundColor: 'white',
      },
      mainContainer: {
        display: 'flex',
        flexDirection: 'column',
      },
      firstContainer: {
        backgroundColor: 'red',
      },
      secondContainer: {
        backgroundColor: 'yellow',
      },
      thirdContainer: {
        height: 200,
        backgroundColor: 'green',
      },
    });

    export default styles;
```

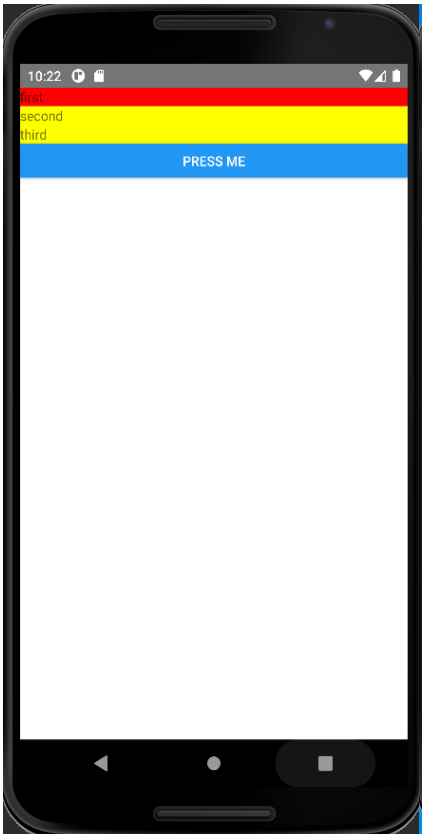


Рисунок 6.1 – Вигляд мобільного застосунку до натиснення кнопки

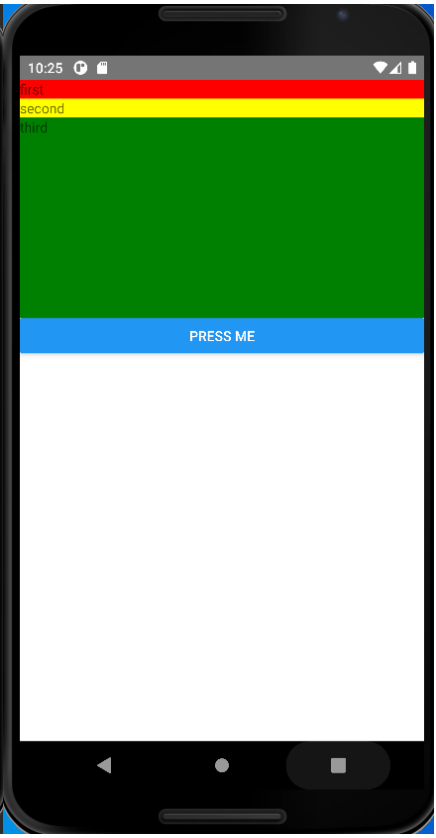


Рисунок 6.2 – Вигляд мобільного застосунку після натиснення кнопки

Властивість flex визначає, як ваші елементи будуть «заповнювати» вільний простір вздовж головної осі. Простір буде розділено відповідно до властивості flex кожного елемента.

Властивість flexDirection контролює напрямок, у якому розміщуються дочірні вузли. Він також називається головною віссю. Поперечна вісь – це вісь, перпендикулярна до головної осі, або вісь, на якій розташовані лінії обгортання:

– row – вирівняти дочірні елементи зліва направо. Якщо увімкнено обтікання, то наступний рядок починатиметься під першим елементом зліва від контейнера.

– column (значення за замовчуванням) – вирівняти дочірні елементи зверху вниз. Якщо увімкнено обтікання, то наступний рядок буде починатися з першого елемента зліва у верхній частині контейнера.

– row-reverse – вирівняти дочірні елементи справа наліво. Якщо увімкнено, то наступний рядок буде починатися під першим елементом праворуч від контейнера.

– column-reverse – вирівняти дочірні елементи знизу вгору. Якщо увімкнено обтікання, то наступний рядок буде починатися ліворуч від першого елемента внизу контейнера.

Властивість justifyContent описує, як вирівняти дочірні елементи за головною віссю їхнього контейнера.[7] Наприклад, ви можете використовувати цю властивість для вирівнювання дочірнього елемента по горизонталі у контейнері зі flexDirection, встановленим у рядку, або по вертикалі у контейнері з flexDirection, встановленим у стовпчику:

– flex-start (значення за замовчуванням) – вирівнює дочірні елементи контейнера за початком головної осі контейнера.

– flex-end – вирівнює дочірні елементи контейнера за кінцем головної осі контейнера.

– center – вирівнює дочірні елементи контейнера по центру головної осі контейнера.

– space-between – вирівняти простір дочірніх елементів вздовж головної осі контейнера, розподіляючи решту простору між ними.

– space-around – рівномірно розміщує дітей вздовж головної осі контейнера, розподіляючи решту простору навколо дітей. Порівняно зі space-between, використання space-around призведе до того, що простір буде розподілено від початку першої дитини до кінця останньої.

– space-evenly –Рівномірно розподіляє простір у контейнері вирівнювання вздовж головної осі. Відстань між кожною парою сусідніх елементів, головним початковим краєм і першим елементом, а також головним кінцевим краєм і останнім елементом, є абсолютно однаковою.

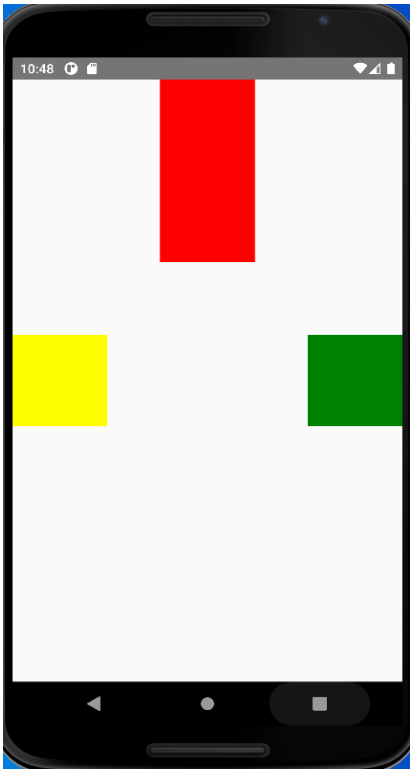


Рисунок 6.3 – Зовнішній вигляд мобільного застосунку

```
import React from 'react';
import { StyleSheet, View, SafeAreaView } from 'react-native';

export default function App() {
  return (
    <SafeAreaView style={styles.mainBlock}>
      <View style={[styles.box, {backgroundColor: 'yellow'}]}></View>
      <View style={[styles.box, {backgroundColor: 'red', height: 200, alignSelf: 'flex-start'}]}></View>
      <View style={[styles.box, {backgroundColor: 'green'}]}></View>
    </SafeAreaView>
```

```
    );
}

const styles = StyleSheet.create({
    mainBlock: {
        flex: 1,
        flexDirection: 'row',
        justifyContent: 'space-between',
        alignItems: 'center'
    },
    box: {
        width: 100,
        height: 100
    }
});
```

Використання декоративних стилів тексту

Далі ми почнемо застосовувати декоративні стилі до тексту. Ми подивимось, як створювати такі ефекти, як підкреслення та закреслення тексту, а також додавання тіней. Ці прийоми можуть урізноманітнити візуальне оформлення застосунків і допомогти текстовим елементам виділятися один на одному.

Ось властивості, які ми розглянемо:

- iOS та Android – lineHeight, textAlign, textDecorationLine, textShadowColor, textShadowOffset та textShadowRadius;
- тільки для Android – textAlignVertical;
- тільки для iOS – letterSpacing, textDecorationColor, textDecorationStyle та writingDirection.

Зверніть увагу, що деякі з властивостей застосовуються лише до тієї чи іншої ОС. Деякі значення, які можна призначити властивостям, також залежать від ОС. Важливо пам’ятати про це, особливо якщо ви покладаєтеся на певний стиль для виділення певного елемента тексту на екрані.

Задання висоти текстових елементів

LineHeight задає висоту елемента Text. На рис. 6.4 і в лістингу показано приклад того, як цей параметр поводиться по-різному на iOS і Android. До елемента Text B застосовано значення lineHeight, рівне 100: висота цього рядка значно більша за висоту інших. Також зверніть увагу, як iOS і Android по-різному позиціонують текст у рядку. На Android текст розташовується внизу рядка.

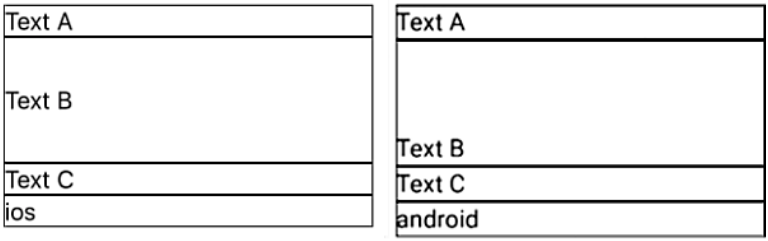


Рисунок 6.4 – Приклад використання lineHeight на iOS та Android

Лістинг: Застосування lineHeight до текстового елемента в iOS та Android

```
import React, { Component } from 'react';
import { Platform, StyleSheet, Text, View} from 'react-native';

export default class App extends Component {
render() {
      return (
        <View style={styles.container}>
          <TextContainer>
            <LeftText>Text A</LeftText>
          </TextContainer>
          <TextContainer>
            <LeftText style={{lineHeight: 100}}>
              Text B
            </LeftText>
          </TextContainer>
          <TextContainer>
            <LeftText>Text C</LeftText>
          </TextContainer>
          <TextContainer>
            <LeftText>{Platform.OS}</LeftText>
          </TextContainer>
        </View>
      );
    }
}

const LeftText = (props) => (
  <Text style={[styles.leftText, props.style]}>
```

```
        {props.children}
    </Text>
);

const TextContainer = (props) => (
    <View style={[styles.textContainer, props.style]}>
        {props.children}
    </View>
);

const styles = StyleSheet.create({
    container: {
        width: 300,
        height: 300,
        margin: 40,
        marginTop: 100
    },
    textContainer: {
        borderWidth: 1
    },
    leftText: {
        fontSize: 20
    }
});
```

Вирівнювання тексту по горизонталі

textAlign визначає, як текст в елементі буде вирівняно по горизонталі. Параметри textAlign мають такі значення: «auto», «center», «right», «left» та «justify» («justify» – лише для iOS).

Підкреслення тексту або додавання ліній крізь текст

Використовуйте властивість textDecorationLine, щоб додати підкреслення або лінію крізь текст. Для textDecorationLine доступні такі варіанти: 'none', 'underline', 'line-through' та 'underline line-through'. Значення за замовчуванням – 'none'. Якщо ви вказуєте «підкреслення через лінію», значення у лапках відокремлюються одним пробілом.

Стилі оформлення тексту (лише для iOS)

iOS підтримує кілька стилів оформлення тексту, яких немає в Android. Перший – textDecorationColor, який дозволяє встановити колір для textDecorationLine. iOS також підтримує стилізацію самого рядка. На Android лінія завжди суцільна, але на iOS textDecorationStyle

дозволяє вказати «суцільний», «подвійний», «пунктирний» і «штриховий». Android ігноруватиме ці додаткові стилі.

Щоб використовувати додаткові стилі оздоблення для iOS, вкажіть їх разом з основним стилем textDecorationLine. Наприклад:

```
{
   textDecorationLine: 'underline',
   textDecorationColor: 'red',
   textDecorationStyle: 'double'

},
textContainer: {
   borderWidth: 1
},
leftText: {
   fontSize: 20
   }
});
```

Додавання тіней до тексту

Ви можете використовувати властивості textShadowColor, textShadowOffset і textShadowRadius для додавання тіні до текстового елемента. Щоб створити тінь, вам потрібно вказати три речі:

- колір;
- зміщення;
- радіус.

Зсув визначає положення тіні відносно компонента, який її відкидає. Радіус в основному визначає, наскільки розмитою буде тінь. Ви можете вказати тінь для тексту таким чином:

```
{
 textShadowColor: 'red',
 textShadowOffset: {width: -2, height: -2},
 textShadowRadius: 4
}
```

Керування міжрядковим інтервалом (лише для iOS)

letterSpacing задає інтервал між символами тексту. Це не те, що ви будете використовувати щодня, але це може створити цікаві візуальні ефекти. Майте на увазі, що він доступний лише для iOS, тому використовуйте його за потреби.[8]

Приклади текстових стилів

На рис. 6.5 показано різні стилі, застосовані до текстових компонентів.

Ось короткий огляд стилів, що використовуються для кожного прикладу на рис. 6:

- A – виділення тексту курсивом за допомогою {fontStyle: 'italic'}.
- B показує оформлення тексту з підкресленням і лінією через текст. Для цього використовується стиль {textDecorationLine: 'underline line-through'}.
- C розширює приклад B, застосовуючи деякі стилі тексту лише для iOS, {textDecorationColor: 'red', textDecorationStyle: 'dotted'}. Зверніть увагу, що ці стилі не впливають на Android.
- D застосовує тінь за допомогою {textShadowColor: 'red', textShadowOffset: {width: -2, height: -2}, textShadowRadius: 4}.
- E використовує тільки для iOS {letterSpacing: 5}, що не впливає на Android.

Текст ios та android стилізовано за допомогою {textAlign: 'center', fontWeight: 'bold'}.

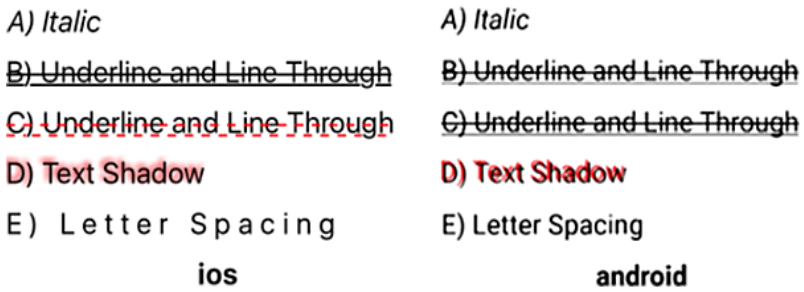


Рисунок 6.5 – Різні приклади стилізації текстових компонентів

Розміри та стилі, що залежать від платформи

Ви бачили, як використовувати функцію Platform.select для вибору шрифтів, доступних тільки на iOS або Android. Ви використовували Platform.select для вибору моноширинного шрифту, який підтримує кожна платформа. Можливо, ви не надто замислювалися про це в той час, але важливо пам’ятати, що ви розробляєте для двох різних платформ. Стилі, що застосовуються до компонента, можуть виглядати або поводитися по-різному в різних ОС або навіть різних версіях iOS і Android.

Ви не пишете код для одного пристрою; ви навіть не кодуєте для однієї ОС. Принадність React Native в тому, що ви використовуєте

JavaScript для створення програм, які можуть працювати як на iOS, так і на Android. Якщо ви переглянете документацію React Native, ви побачите безліч компонентів із суфіксами IOS або Android, таких як ProgressBarAndroid, ProgressViewIOS та ToolbarAndroid, тому не дивно, що стилі також можуть залежати від платформи.

Можливо, ви не помітили, що ви ніколи не вказували розмір пікселів для чого-небудь, наприклад, width: 300 vs width: '300px'. Це пов'язано з тим, що навіть поняття розміру в операційних системах iOS та Android відрізняється.[9]

Пікселі, точки та DP

Розмір може бути заплутаною темою, але важливо пам'ятати, якщо вам потрібно бути абсолютно точним при розміщенні компонентів на екрані. Навіть якщо ви не намагаєтеся створити макет із високою точністю, буде корисно зрозуміти концепції на випадок, якщо ви зіткнетеся з невеликими невідповідностями в макетах з одного пристрою на інший.

Почнемо із самого початку і визначимо піксель. Піксель – це найменша одиниця програмованого кольору на дисплеї. Піксель зазвичай складається з компонентів червоного, зеленого та синього (RGB) кольорів. Керуючи інтенсивністю кожного значення RGB, піксель випромінює колір, що ви бачите. Піксель нічого вам не каже, поки ви не почнете дивитися на фізичні властивості дисплея: розмір екрану, роздільна здатність та кількість точок на дюйм.

Розмір екрана – це розмір екрана по діагоналі від одного кута до іншого. Наприклад, вихідний розмір екрану iPhone становив 3,5 дюйми, а розмір екрану iPhone X – 5,8 дюйми. Хоча iPhone X значно більше, розмір нічого не означає, поки ви не зрозумієте, скільки пікселів уміщається на цьому розмірі екрану.

Роздільна здатність – це кількість пікселів на дисплеї, яка найчастіше виражається як кількість пікселів по ширині та висоті пристрою. Оригінальний iPhone мав дозвіл 320×480, а iPhone X – 1125×2436.

Потім розмір екрану та роздільну здатність можна використовувати для розрахунку щільності пікселів: пікселів на дюйм (PPI). Ви часто бачитимете, що це виражається в точках на дюйм (DPI), що є пережитком зі світу друку, де точка кольору була надрукована на сторінці. PPI і DPI часто використовуються як взаємозамінні, хоча це не зовсім правильно, тому, якщо ви бачите, що DPI використовується щодо екрана, знайте, що насправді обговорюється PPI.

PPI дає вам міру різкості зображення. Уявіть, якби два екрани мали однакову роздільну здатність 320 × 480 (половина VGA). Як виглядa-

тиме те саме зображення на 3,5-дюймовому дисплеї iPhone у порівнянні з 17-дюймовим дисплеєм монітора HVGA? Те саме зображення буде виглядати набагато чіткіше на iPhone, тому що у нього 163 PPI в порівнянні з ЕПТ-монітором, у якого 34 PPI. Ви можете розмістити майже в п'ять разів більше інформації у тому ж фізичному просторі на оригінальному iPhone.

Наближення тіней на пристрої Android з урахуванням висоти

Як досягти такого ж ефекту на пристроях Android? Правда в тому, що ви не можете цього зробити. Ви можете використовувати стиль висоти Android, щоб вплинути на z-порядок компонентів. Якщо два або більше компоненти займають один і той же простір, ви можете вирішити, який з них повинен бути попереду, задавши йому велику висоту і, отже, більший z-індекс, що створить невелику тінь, але це не так впадає у вічі, як ефекти тіней, які можна отримати на iOS. Зверніть увагу, що це стосується тільки Android, тому що iOS не підтримує стиль піднесення і проігнорує його, якщо він вказаний.

Тим не менш, давайте подивимося піднесення у дії. Для цього ви створите компонент View із трьома блоками, кожен із яких розташований абсолютно. Ви поставите їм три різні позначки – 1, 2 і 3 – а потім поміняєте призначення позначок на протилежне і подивіться, як це вплине на компонування. На рис. 1 показані результати цих виправлень по висоті.

У табл. 1 показані абсолютні положення та висоти, що використовуються для кожної групи скриньок. Зверніть увагу, що нічого не змінилося, крім висоти, призначеної для кожного з полів. iOS ігнорує стиль і завжди відображає блок C поверх блоку B, а блок B поверх блоку A. Але Android дотримується стилю і змінює порядок відображення блоків, тому блок A тепер знаходиться поверх блоку B, а блок B знаходиться поверх блоку C.

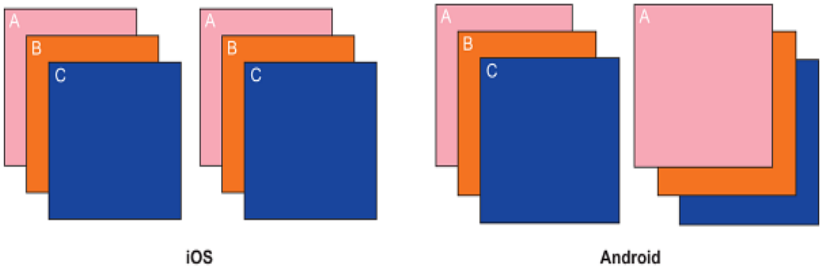


Рисунок 6.6 – Приклади використання стилю висоти на iOS та Android. В iOS висота ігнорується; всі компоненти зберігають той самий z-

порядок, тому будь-який компонент, який знаходиться останнім у макеті, знаходиться зверху. В Android використовується висота та змінюється z-порядок; у другому прикладі, де призначення висот змінені протилежні, А знаходиться зверху.

Завдання

1. Створити 3 інпути R, G, B, які набуватимуть значення від 0 до 255 і змінюватимуть колір блоку згідно з RGB.
2. Зробити таблицю розміром 3 на 3 елементи з використанням flexbox.
3. Додати можливість змінювати висоту і ширину блоку кнопками більше/ менше.

Контрольні питання

1. Навіщо використовувати StyleSheet.create() у React Native?
2. Як використовується flexbox?
3. Як зробити динамічний стиль?
4. Що таке умовний стиль?
5. Як застосувати кілька стилів до компонента?

ЛАБОРАТОРНА РОБОТА №7
«ВИКОРИСТАННЯ БІБЛІОТЕКИ NATIVEBASE У МОБІЛЬНИХ ЗАСТОСУНКАХ REACT NATIVE»

Встановлення бібліотеки NativeBase

NativeBase – це бібліотека компонентів для React та React Native, орієнтована на мобільні пристрої. Версія 3.0 містить підтримку утиліт та майже 40 компонентів, які сумісні з Android, iOS та Web.

NativeBase підтримується в Expo або React Native CLI, ініційованих додатками. Вебпідтримка стала можливою завдяки react-native-web.

Expo допомагає створювати універсальні (для iOS, Android та Web) React Native додатки без конфігурації збірки. Створіть новий проєкт за допомогою Expo CLI з шаблоном NativeBase:

expo init my-app --template @native-base/expo-template – простий шаблон JavaScript;

expo init my-app --template @native-base/expo-template-typescript – шаблон TypeScript.

Якщо проєкт вже створено, то потрібно встановити залежності, попередньо перейшовши у папку проєкту:

npm install native-base

expo install react-native-svg@12.1.1

expo install react-native-safe-area-context@3.3.2

Створення нового проєкту за допомогою ReactNative CLI з шаблоном NativeBase:

npx react-native init MyApp --template @native-base/react-native-template – простий шаблон JavaScript;

npx react-native init MyApp --template @native-base/react-native-template-typescript – шаблон TypeScript.

Якщо проєкт вже створено, то потрібно встановити залежності, попередньо перейшовши у папку проєкту:

npm install native-base react-native-svg@12.1.1 react-native-safe-area-context@3.3.2

Об’єкт NativeBaseProvider

NativeBaseProvider – це компонент, який робить тему доступною у всьому застосунку. Він використовує контекстний API React. Додати NativeBaseProvider у корінь застосунку можна наступним чином:

```
import React from "react";
```

```
import { NativeBaseProvider, Text, Box } from "native-base";

export default function App() {
  return (
    <NativeBaseProvider>
      <Box flex={1} bg="#fff" alignItems="center" justifyContent="center">
        <Text>Open up App.js to start working on your app!</Text>
      </Box>
    </NativeBaseProvider>
  );
}
```

Якщо вам потрібно налаштувати тему за замовчуванням відповідно до ваших вимог до дизайну, ви можете розширити тему з native-base.

У NativeBase 3.0 передбачено функцію extendTheme, яка глибоко інтегрує типову тему з вашими налаштуваннями.

```
import { extendTheme, NativeBaseProvider } from "native-base";

const newColorTheme = {
  brand: {
    900: "#8287af",
    800: "#7c83db",
    700: "#b3bef6",
  },
};
const theme = extendTheme({ colors: newColorTheme });

function App() {
  return (
    <NativeBaseProvider theme={theme}>
      <App />
    </NativeBaseProvider>
  );
}
```

Контейнерні компоненти

Box

Це загальний компонент для низькорівневої верстки. Він схожий на div в HTML. (рис. 7.1)

```
import { Box } from "native-base"
```

```
export default function App() {
  return <Box bg="primary.400" p={4} _text={{
    fontSize: "md",
    fontWeight: "bold",
    color: "white"
  }}>
    This is a Box
  </Box>;
}
```

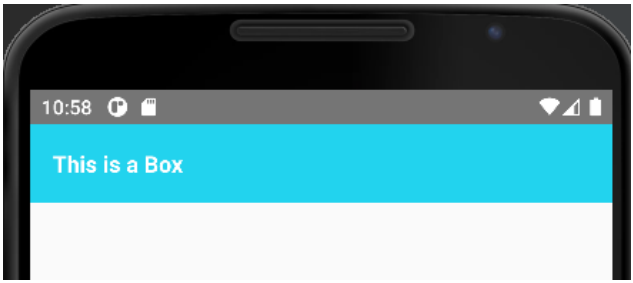


Рисунок 7.1 – Компонента коробка

Center

Center вирівнює вміст по центру всередині себе. Це компонент макета. (рис. 7.2)

```
import React from "react"
import { NativeBaseProvider, Center, Text } from "native-base"

export default function App() {
  return <NativeBaseProvider>
    <Center bg="primary.400" _text={{color: "white", fontWeight: "bold" }}
      height={200} width={{
        base: 200,
        lg: 400
      }}>
      This is the Center
    </Center>
  </NativeBaseProvider>
}
```

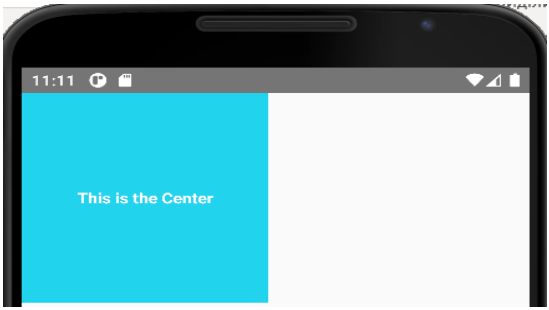


Рисунок 7.2 – Компонента Center

Stack

Stack вирівнює елементи по вертикалі або горизонталі на основі реквізиту **direction**.

```
import React from "react"
import { NativeBaseProvider, Center, Stack, Button, Heading } from "native-base"

export default function App() {
  const [direction, setDirection] = React.useState("column")
  return <NativeBaseProvider>
    <Stack space={3} alignItems="center">
    <Heading>Stack - {direction === "row" ? "Row" : "Column"}</Heading>
    <Stack direction={direction} space={3} mb={3} alignItems="center">
      <Center size={16} bg="primary.400" rounded="md" _text={{
        color: "white"
      }} shadow={3}>
        Box 1
      </Center>
      <Center bg="secondary.400" size={16} rounded="md" _text={{
        color: "white"
      }} shadow={3}>
        Box 2
      </Center>
      <Center size={16} bg="emerald.400" rounded="md" _text={{
        color: "white"
      }} shadow={3}>
```

```
            Box 3
          </Center>
        </Stack>
        <Button onPress={() => setDirection(direction === "row" ? "column" : "row")}>
          Change Stack Direction
        </Button>
      </Stack>
      </NativeBaseProvider>
  }
```

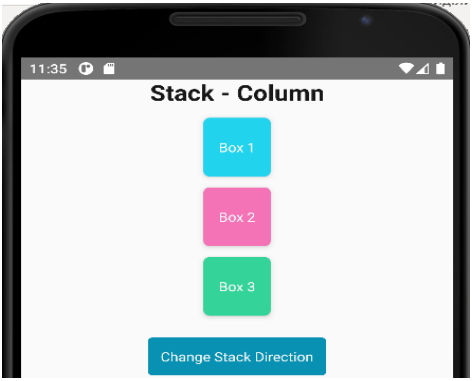


Рисунок 7.3 – Компонента Stack у стовпчик

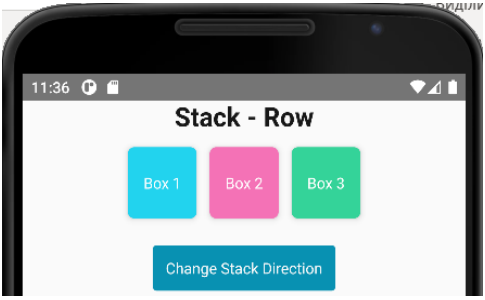


Рисунок 7.4 – Компонента Stack в рядок

HStack / Row

HStack вирівнює елементи по горизонталі. Row також є псевдонімом для HStack.

```
import React from "react"
```

```
import { NativeBaseProvider, Center, Stack, HStack } from "native-base"

export default function App() {
  return <NativeBaseProvider>
    <Stack space={3} alignItems="center">
      <HStack space={3} alignItems="center">
        <Center size={16} bg="primary.400" rounded="md" _text={{
          color: "white"
        }} shadow={3}>
          Box 1
        </Center>
        <Center bg="secondary.400" size={16} rounded="md" _text={{
          color: "white"
        }} shadow={3}>
          Box 2
        </Center>
        <Center size={16} bg="emerald.400" rounded="md" _text={{
          color: "white"
        }} shadow={3}>
          Box 3
        </Center>
      </HStack>
    </Stack>
  </NativeBaseProvider>
}
```

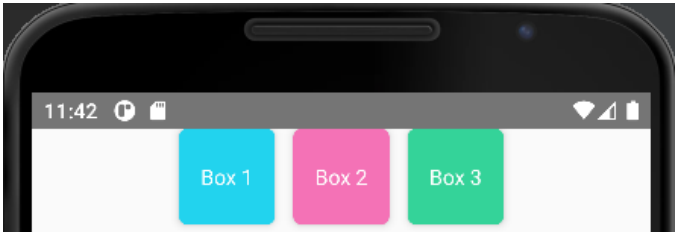


Рисунок 7.5 – HStack

VStack / Column

VStack вирівнює елементи по вертикалі. Column також є псевдонімом для VStack.

```
import React from "react"
```

```
import { NativeBaseProvider, Center, Stack, VStack, Heading } from "native-base"

export default function App() {
  return <NativeBaseProvider>
    <VStack space={4} alignItems="center">
      <Heading>VStack</Heading>
      <Center size={16} bg="primary.400" rounded="md" _text={{
        color: "white"
      }} shadow={3}>
        Box 1
      </Center>
      <Center bg="secondary.400" size={16} rounded="md" _text={{
        color: "white"
      }} shadow={3}>
        Box 2
      </Center>
      <Center size={16} bg="emerald.400" rounded="md" _text={{
        color: "white"
      }} shadow={3}>
        Box 3
      </Center>
    </VStack>
  </NativeBaseProvider>
}
```

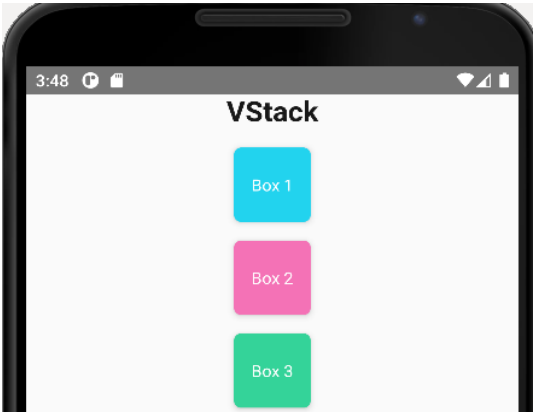


Рисунок 7.6 – VStack

ZStack

ZStack вирівнює елементи за віссю z.

```
import React from "react"
import { NativeBaseProvider, Box, ZStack } from "native-base"

export default function App() {
  return <NativeBaseProvider>
    <Box h="40" mx="auto">
      <ZStack mt={3} ml={-20}>
        <Box bg="primary.400" size={20} rounded="lg" shadow={3}
/>
        <Box bg="secondary.400" mt={5} ml={5} size={20}
rounded="lg" shadow={5} />
        <Box bg="emerald.400" mt={10} ml={10} size={20}
rounded="lg" shadow={7} />
      </ZStack>
    </Box>
  </NativeBaseProvider>
}
```

Рисунок 7.7 – ZStack

Ви можете передати alignItems="center" justifyContent="center" для вертикального та горизонтального центрування дочірніх елементів.

```
import React from "react"
import { NativeBaseProvider, Box, ZStack } from "native-base"

export default function App() {
  return <NativeBaseProvider>
```

```
        <ZStack alignItems="center" justifyContent="center">
          <Box bg="primary.400" size={64} rounded="lg" />
          <Box bg="secondary.400" size={48} rounded="lg" shadow={8}
/>
          <Box bg="emerald.400" size={32} rounded="lg" shadow={8}
/>
        </ZStack>
      </NativeBaseProvider>
    }
```

Рисунок 7.8 – Центрування стаку

Візуальні елементи

Button

Компонент Button використовується для запуску дії або події. (рис. 7.9)

```
import React from "react"
import { NativeBaseProvider, Button, Stack } from "native-base"

export default function App() {
  return <NativeBaseProvider>
    <Stack direction={{
      base: "column",
      md: "row"
    }} space={2} mx={{
      base: "auto",
      md: 0
    }}>
      <Button size="sm" onPress={() => console.log("hello world")}>
        PRIMARY
      </Button>
```

```
            <Button size="sm" colorScheme="secondary" onPress={() =>
console.log("hello world")}>
                SECONDARY
            </Button>
            <Button size="sm" isDisabled onPress={() => console.log("hello
world")}>
                DISABLED
            </Button>
        </Stack>
    </NativeBaseProvider>
}
```

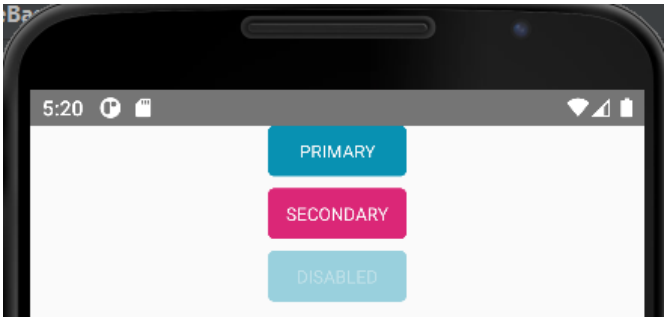


Рисунок 7.9 – Кнопка

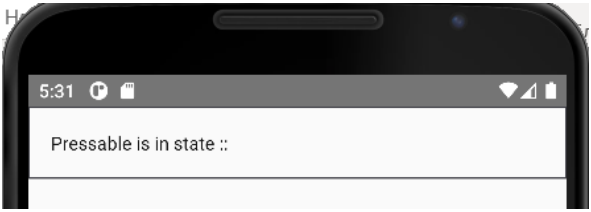
Pressable

Pressable – примітив нижчого рівня, якщо вам потрібна більша гнучкість, ніж кнопка, і доступ до подій наведення, натискання та фокусування.

```
import React from "react"
import { NativeBaseProvider, Pressable, Text } from "native-base"

export default function App() {
    return <NativeBaseProvider>
        <Pressable p={4} borderWidth={1} _light={{
            borderColor: "dark.200"
        }} _dark={{
            borderColor: "dark.600"
        }}>
            {({
                isHovered,
                isFocused,
```

```
                isPressed
              }) => {
                return <Text>
                  Pressable is in state{" :: "}
                  {isPressed ? "pressed" : isHovered ? "hovered" : isFocused
? "focused" : ""}
                </Text>;
              }}
            </Pressable>
          </NativeBaseProvider>
      }
```



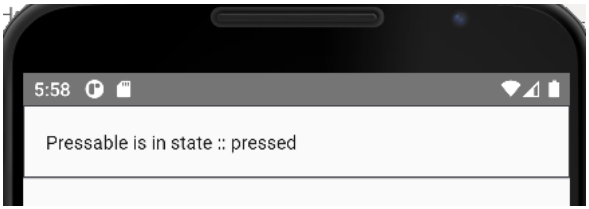


Рисунок 7.10 – Pressable в різних станах

Input

Компонент Input – це компонент, який використовується для отримання даних, введених користувачем у текстове поле.

```
import React from "react"
import { NativeBaseProvider, Input, Button } from "native-base"

export default function App() {
  const [show, setShow] = React.useState(false);
  const handleClick = () => setShow(!show);

  return <NativeBaseProvider>
```

```
        <Input type={show ? "text" : "password"}
InputRightElement={<Button ml={1} roundedLeft={0}
roundedRight="md" onPress={handleClick}>
            {show ? "Hide" : "Show"}
          </Button>} placeholder="Password" />
      </NativeBaseProvider>
  }
```

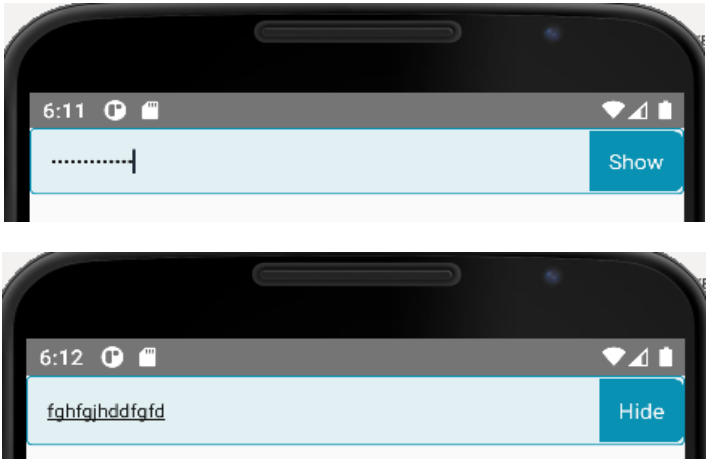


Рисунок 7.11 – Input в різних станах

Завдання

1. Створіть форму авторизації в React Native за допомогою компонентів бібліотеки Native Base, використовуючи стилі.
2. Додайте спливаючі повідомлення «Авторизація пройшла успішно» при успішній авторизації (задайте як константу деякий логін і пароль або словник логінів і паролів) та «Помилка авторизації» при неправильному введенні.

Контрольні питання

1. Яке призначення компоненту NativeBaseProvider?
2. Як створити власну тему у NativeBase та підключити її до мобільного застосунку?
3. Які контейнерні компоненти бібліотеки NativeBase можна використати для впорядкування візуальних елементів? Чим вони відрізняються між собою?
4. Як налаштовуються стилі візуальних елементів бібліотеки NativeBase?

ЛАБОРАТОРНА РОБОТА №8
«РЕАЛІЗАЦІЯ НАТИВНОЇ АНІМАЦІЇ У ЗАСТОСУНКАХ REACT NATIVE»

Теоретичні відомості

Нативні анімації React надають додатковий ефект та покращують користувацький досвід у додатку. Анімації передають фізично правдоподібний рух у вашому інтерфейсі.

У React Native є два типи додаткових систем анімації. Це

Animated: Анімований API використовується для інтерактивного керування конкретними значеннями. Він фокусується на декларативних зв'язках між входами та виходами. Він має методи запуску і зупинки для керування виконанням анімації в залежності від часу. Animated експортує чотири різні анімовані компоненти: View, Text, Image і ScrollView. Ми також можемо створити власний анімований компонент за допомогою Animated.createAnimatedComponent().[9]

LayoutAnimated: LayoutAnimated використовується для анімування глобальних транзакцій компонування.

Методи анімації

Методи	Опис
Animated.timing()	Анімує значення в часі, використовуючи різні криві згладжування або власну функцію.
Animated.event()	Зіставляє події безпосередньо з анімованими значеннями.
Animated.spring()	Анімує значення, відстежує швидкість зміни стану для створення плавного руху при оновленні значення toValue.
Animated.decay()	Починає анімацію з початковою швидкістю і поступово сповільнюється до зупинки.

Приклад 1

```
import React, { Component } from 'react';
import {StyleSheet, AppRegistry,Text, View,Animated,Easing} from 'react-native';

export default class App extends Component {
  constructor () {
    super()
```

```
    this.spinValue = new Animated.Value(0) //оголосити spinValue як новий Animated.Value і передати в ньому 0 (нуль)
  }

  componentDidMount () {
    this.spin()
  }

  //створити метод spin і викликати його з componentDidMount
  spin() {
    this.spinValue.setValue(0) // встановити spinValue рівним 0
    Animated.timing(      // Виклик методу Animated.timing() отримує два аргументи:
      this.spinValue, // значення
      {          // і об'єкт конфігурації
        toValue: 1, // і встановивши spinValue рівним 1
        duration: 4000, // протягом 4000 мілісекунд
        easing: Easing.linear
      }
    ).start(() => this.spin())
  }
  render() {
    const spin = this.spinValue.interpolate({
      inputRange: [0, 1],
      outputRange: ['0deg', '360deg']
    })
    return (
      <View style={styles.container}>
        <Animated.Image
          style={{
            width: 227,
            height: 200,
            transform: [{rotate: spin}] }}
          source={require('./react_logo.png')}
        />
      </View>
    )
  }
}
const styles = StyleSheet.create({
  container: {
```

```
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  }
})
```

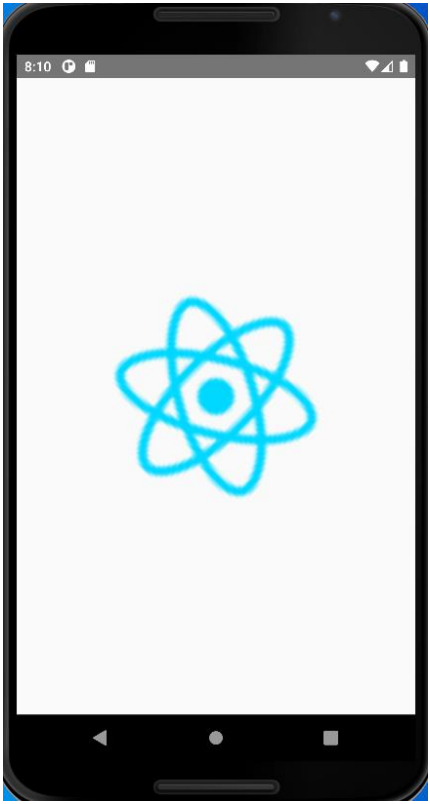


Рисунок 8.1 – Приклад анімації обертання

Приклад 2

```
import React, { Component } from 'react';
import {StyleSheet, AppRegistry,Text, View,Animated,Easing} from 'react-native';

export default class App extends Component {
  constructor() {
```

```
    super()
    this.animatedValue = new Animated.Value(0)
}

componentDidMount() {
    this.animate()
} //метод animate викликається з componentDidMount
animate() {
    this.animatedValue.setValue(0)
    Animated.timing(
        this.animatedValue,
        {
            toValue: 1,
            duration: 2000,
            easing: Easing.linear
        }
    ).start(() => this.animate())
}

render() {
    const marginLeft = this.animatedValue.interpolate({
        inputRange: [0, 1],
        outputRange: [0, 300]
    })
    const opacity = this.animatedValue.interpolate({
        inputRange: [0, 0.5, 1],
        outputRange: [0, 1, 0]
    })
    const movingMargin = this.animatedValue.interpolate({
        inputRange: [0, 0.5, 1],
        outputRange: [0, 300, 0]
    })
    const textSize = this.animatedValue.interpolate({
        inputRange: [0, 0.5, 1],
        outputRange: [18, 32, 18]
    })
    const rotateX = this.animatedValue.interpolate({
        inputRange: [0, 0.5, 1],
        outputRange: ['0deg', '180deg', '0deg']
    })
```

```
        return (
            <View style={styles.container}>
                <Animated.View // повертає Animated.View
                    style={{
                        marginLeft,
                        height: 30,
                        width: 40,
                        backgroundColor: 'red'}} />
                <Animated.View
                    style={{
                        opacity,
                        marginTop: 10,
                        height: 30,
                        width: 40,
                        backgroundColor: 'blue'}} />
                <Animated.View
                    style={{
                        marginLeft: movingMargin,
                        marginTop: 10,
                        height: 30,
                        width: 40,
                        backgroundColor: 'orange'}} />
                <Animated.Text // returns Animated.Text
                    style={{
                        fontSize: textSize,
                        marginTop: 10,
                        color: 'green'}} >
                    Animated Text!
                </Animated.Text>
                <Animated.View
                    style={{
                        transform: [{rotateX}],
                        marginTop: 50,
                        height: 30,
                        width: 40,
                        backgroundColor: 'black'}}>
                    <Text style={{color: 'white'}}>Hello from
TransformX</Text>
                </Animated.View>
            </View>
        )
```

```
    }
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        paddingTop: 150
    }
})
```

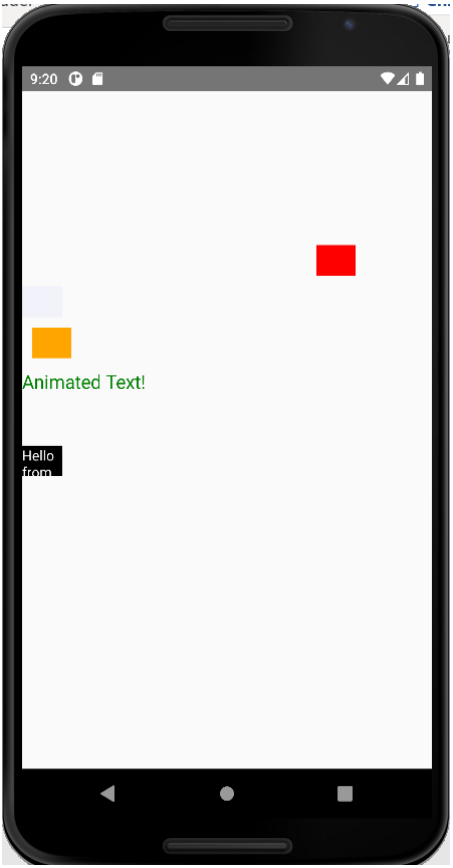


Рисунок 8.2 – Приклад анімації лінійної

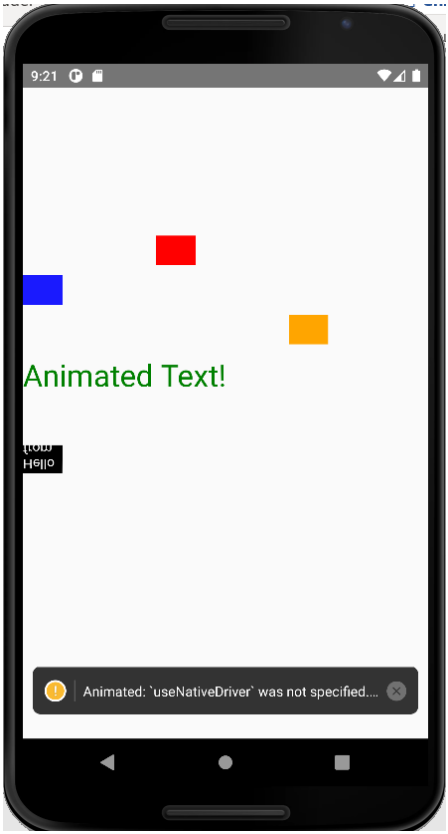


Рисунок 8.3 – Приклад анімації лінійної з іншої сторони

Обробка жестів та інших подій

Жести, такі як панорамування або прокрутка, та інші події можуть бути відображені безпосередньо в анімовані значення за допомогою функції Animated.event(). Це робиться за допомогою структурованого синтаксису відображення, щоб значення можна було витягувати зі складних об'єктів подій. Перший рівень – це масив, який дозволяє відображати декілька аргументів, і цей масив містить вкладені об'єкти.[10]

Наприклад, працюючи із жестами горизонтальної прокрутки, ви можете зробити наступне, щоб зіставити event.nativeEvent.contentOffset.x зі scrollX (анімованим значенням):

onScroll={Animated.event(

```
    // scrollX = e.nativeEvent.contentOffset.x
    [{nativeEvent: {
        contentOffset: {
          x: scrollX
        }
      }
    }]
  )}
```

Приклад

```
import React, { useRef } from "react";
import {
  SafeAreaView,
  ScrollView,
  Text,
  StyleSheet,
  View,
  ImageBackground,
  Animated,
  useWindowDimensions
} from "react-native";

const images = new Array(6).fill('https://images.unsplash.com/photo-
1556740749-887f6717d7e4');

const App = () => {
  const scrollX = useRef(new Animated.Value(0)).current;

  const { width: windowWidth } = useWindowDimensions();

  return (
    <SafeAreaView style={styles.container}>
      <View style={styles.scrollContainer}>
        <ScrollView
          horizontal={true}
          style={styles.scrollViewStyle}
          pagingEnabled
          showsHorizontalScrollIndicator={false}
          onScroll={Animated.event([
            {
              nativeEvent: {
                contentOffset: {
```

```
                x: scrollX
              }
            }
          }
        ])}
        scrollEventThrottle={1}
      >
        {images.map((image, imageIndex) => {
          return (
            <View
              style={{ width: windowWidth, height: 250 }}
              key={imageIndex}
            >
              <ImageBackground source={{ uri: image }}
style={styles.card}>
                <View style={styles.textContainer}>
                  <Text style={styles.infoText}>
                    {"Image - " + imageIndex}
                  </Text>
                </View>
              </ImageBackground>
            </View>
          );
        })}
      </ScrollView>
      <View style={styles.indicatorContainer}>
        {images.map((image, imageIndex) => {
          const width = scrollX.interpolate({
            inputRange: [
              windowWidth * (imageIndex - 1),
              windowWidth * imageIndex,
              windowWidth * (imageIndex + 1)
            ],
            outputRange: [8, 16, 8],
            extrapolate: "clamp"
          });
          return (
            <Animated.View
              key={imageIndex}
              style={[styles.normalDot, { width }]}
            />
```

```
            );
           })}
         </View>
       </View>
     </SafeAreaView>
   );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: "center",
    justifyContent: "center"
  },
  scrollContainer: {
    height: 300,
    alignItems: "center",
    justifyContent: "center"
  },
  card: {
    flex: 1,
    marginVertical: 4,
    marginHorizontal: 16,
    borderRadius: 5,
    overflow: "hidden",
    alignItems: "center",
    justifyContent: "center"
  },
  textContainer: {
    backgroundColor: "rgba(0,0,0, 0.7)",
    paddingHorizontal: 24,
    paddingVertical: 8,
    borderRadius: 5
  },
  infoText: {
    color: "white",
    fontSize: 16,
    fontWeight: "bold"
  },
  normalDot: {
    height: 8,
```

```
    width: 8,
    borderRadius: 4,
    backgroundColor: "silver",
    marginHorizontal: 4
  },
  indicatorContainer: {
    flexDirection: "row",
    alignItems: "center",
    justifyContent: "center"
  }
});

export default App;
```

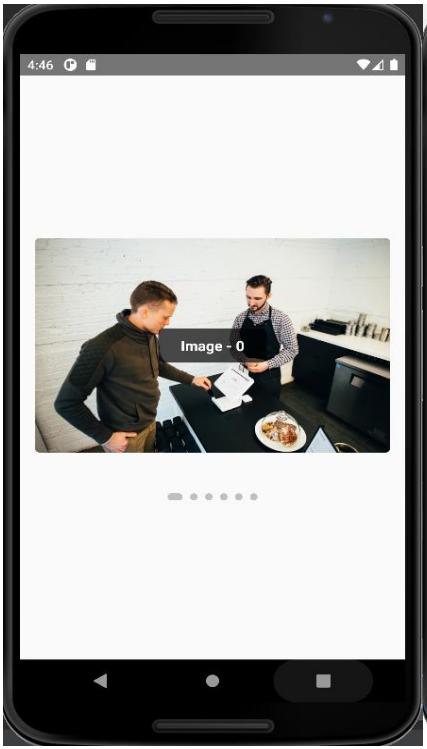


Рисунок 8.4 – Початковий стан жесту

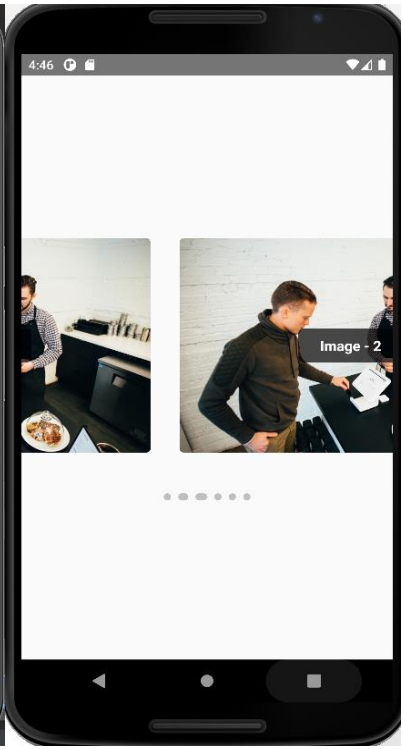


Рисунок 8.5 – Середній стан жесту

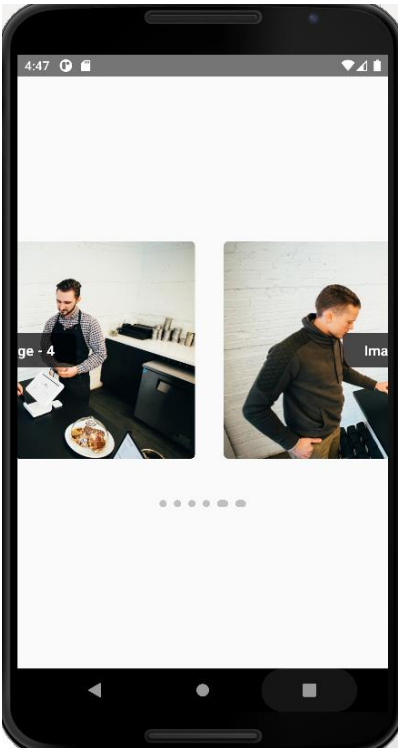


Рисунок 8.6 – Кінцевий результат жесту

Завдання

1. Створіть анімоване вікно привітання (компоненти та їх значення на власний розсуд).
2. Створіть анімацію для слайдера з різними картинками (можна користатись кодом з останнього прикладу).

Контрольні питання

1. Які компоненти React Native можна зробити анімованими?
2. Якими є основні дії при реалізації анімації?
3. У чому полягає призначення методу timing об’єкта Animating?
4. У чому особливості використання інтерполяції у реалізації анімації у застосунках React Native?

ЛАБОРАТОРНА РОБОТА №9
«РЕАЛІЗАЦІЯ НАВІГАЦІЇ МІЖ ДЕКІЛЬКОМА ЕКРАНАМИ У ЗАСТОСУНКУ REACT NATIVE»

Теоретичні відомості

Перехід між екранами

Мобільні додатки рідко складаються з одного екрана. Керування відображенням і переходом між кількома екранами зазвичай здійснюється за допомогою так званого навігатора.

Цей посібник охоплює різні компоненти навігації, доступні в React Native. Якщо ви тільки починаєте працювати з навігацією, вам, ймовірно, варто скористатися React Navigation. React Navigation надає просте рішення для навігації, з можливістю представляти загальну навігацію стеком та шаблони навігації з вкладками як на Android, так і на iOS.

Якщо ви інтегруєте React Native у додаток, який вже керує навігацією нативно, або шукаєте альтернативу React Navigation, наступна бібліотека надає нативну навігацію на обох платформах: react-native-navigation.

React Navigation

Рішення спільноти для навігації – це окрема бібліотека, яка дозволяє розробникам налаштовувати екрани додатку за допомогою декількох рядків коду.[11]

Встановлення та налаштування

Спочатку вам потрібно встановити їх у вашому проєкті:

```
npm install @react-navigation/native @react-navigation/native-stack
```

Далі встановіть необхідні однорангові залежності. Вам потрібно запускати різні команди залежно від того, чи є ваш проєкт керованим Expo або чистим React Native проєктом.

Якщо у вас проєкт під керуванням Expo, встановіть залежності за допомогою expo:

```
npx expo install react-native-screens react-native-safe-area-context
```

Якщо у вас чистий React Native проєкт, встановіть залежності за допомогою npm:

```
npm install react-native-screens react-native-safe-area-context
```

Тепер вам потрібно обернути весь додаток у NavigationContainer. Зазвичай ви робите це у вхідному файлі, наприклад, index.js або App.js:

```
import * as React from 'react';
import {NavigationContainer} from '@react-navigation/native';

const App = () => {
  return (
    <NavigationContainer>
      {/* Rest of your app code */}
    </NavigationContainer>
  );
};

export default App;
```

Тепер ви готові створити та запустити свій додаток на пристрої/симуляторі.

Використання

Тепер ви можете створити додаток з домашнім екраном і екраном профілю:

```
import * as React from 'react';
import { Text, Button } from 'react-native';
import {NavigationContainer} from '@react-navigation/native';
import {createNativeStackNavigator} from '@react-navigation/native-stack';

const Stack = createNativeStackNavigator();
```

У цьому прикладі за допомогою компонента Stack.Screen визначено 2 екрани (Home і Profile). Аналогічно, ви можете визначити стільки екранів, скільки вам потрібно.

Ви можете встановити такі параметри, як заголовок екрана для кожного екрана, у пропорі опцій Stack.Screen.

Кожен екран отримує проп компонента, який є React-компонентом. Ці компоненти отримують проп під назвою navigation, який має різні методи для зв’язку з іншими екранами. Наприклад, ви можете використовувати navigation.navigate для переходу на екран профілю:

```
const HomeScreen = ({navigation}) => {
  return (
    <Button
```

```
      title="Go to Jane's profile"
      onPress={() =>
        navigation.navigate('Profile', {name: 'Jane'})
      }
    />
  );
};

const ProfileScreen = ({navigation, route}) => {
  return <Text>This is {route.params.name}'s profile</Text>;
};
```

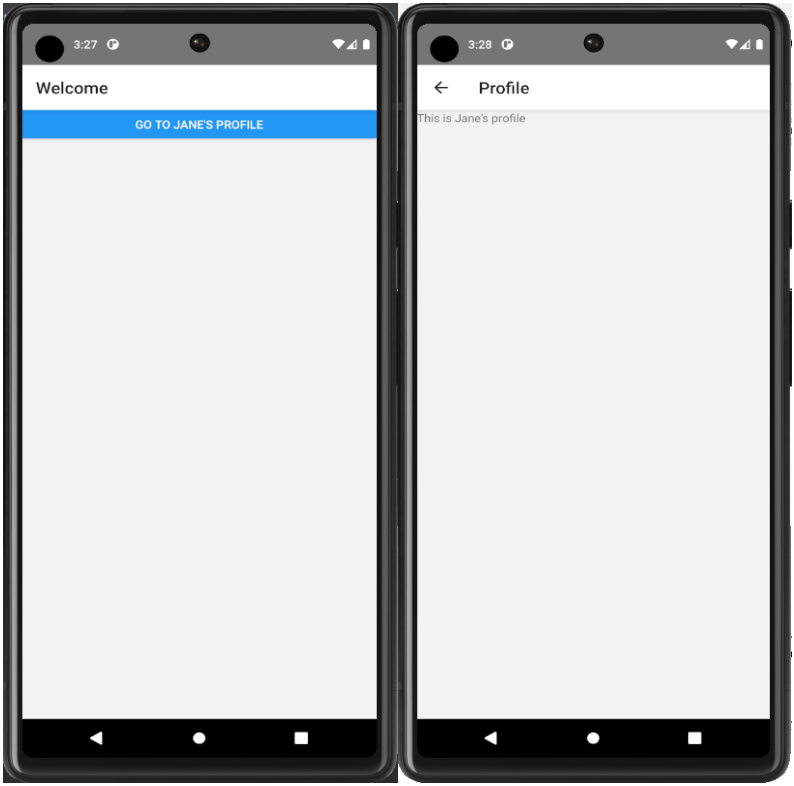


Рисунок 9.1 – Використання навігації в застосунку

Цей навігатор нативного стеку використовує нативні API: UINavigationController на iOS і Fragment на Android, тому навігація, створена за допомогою createNativeStackNavigator, поводитиметься так само і матиме такі ж характеристики продуктивності, як і програми, створені на основі цих API.

Налаштування навігатора

Вся конфігурація маршруту задається як пропси для нашого навігатора. Ми не передали жодних пропсів нашому навігатору, тому він просто використовує конфігурацію за замовчуванням.

Давайте додамо другий екран до нашого рідного навігатора стека і налаштуємо відображення головного екрана першим:

```
function DetailsScreen() {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center'
}}>
      <Text>Details Screen</Text>
    </View>
  );
}

const Stack = createNativeStackNavigator();

function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Home">
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="Details" component={DetailsScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

Рисунок 9.2 – Створений додатковий екран

Тепер у нашому стеку є два маршрути, маршрут «Додому» і маршрут «Деталі». Маршрут можна вказати за допомогою компонента Screen. Компонент Screen приймає проп name, який відповідає назві маршруту, який ми будемо використовувати для навігації, і проп component, який відповідає компоненту, який він буде рендерити.

Тут маршрут Home відповідає компоненту HomeScreen, а маршрут Details відповідає компоненту DetailsScreen. Початковим маршрутом для стека є маршрут Home. Спробуйте змінити його на Details і перезавантажте додаток (швидке оновлення React Native не оновить зміни з initialRouteName, як ви могли б очікувати), помітьте, що тепер ви побачите екран Details. Потім змініть його назад на Home і перезавантажте ще раз.

Вказівка параметрів

На кожному екрані навігатора можна вказати деякі параметри навігатора, наприклад, назву, яку буде показано у заголовку. Ці параметри можна передати у пропсі options для кожного компонента екрана:

```
<Stack.Screen
  name="Home"
  component={HomeScreen}}
  options={{ title: 'Overview' }}
/>
```

Іноді нам потрібно вказати однакові параметри для всіх екранів навігатора. Для цього ми можемо передати навігатору проп screenOptions.

Передача додаткових пропсів

Іноді нам може знадобитися передати додаткові пропси до екрана. Ми можемо зробити це у 2 способи:

1. Використовувати контекст React і обернути навігатор контекстним провайдером для передачі даних на екран (рекомендовано).
2. Використовувати зворотній виклик рендерингу для екрану замість того, щоб вказувати проп компонента:

```
<Stack.Screen name="Home">
  {(props) => <HomeScreen {...props} extraData={someData} />}
</Stack.Screen>
```

За замовчуванням React Navigation застосовує оптимізацію до компонентів екрану, щоб запобігти зайвому рендерингу. Використання зворотного виклику рендерингу прибирає ці оптимізації. Тому, якщо ви використовуєте зворотний виклик рендерингу, вам потрібно переконатися, що ви використовуєте React.memo або React.PureComponent для ваших екранних компонентів, щоб уникнути проблем з продуктивністю.

Переміщення між екранами

До цього ми визначили навігатор стека з двома маршрутами (Home і Details), але ми не дізналися, як дозволити користувачеві перейти з Home до Details (хоча ми навчилися змінювати початковий маршрут в нашому коді, але змушувати наших користувачів клонувати наш репозиторій і змінювати маршрут в нашому коді, щоб побачити інший

екран – це, мабуть, один із найгірших користувацьких досвідів, який можна собі уявити).

Якби це був веб-браузер, ми могли б написати щось подібне:

```
<a href="details.html">Перейти до деталей</a>
```

Інший спосіб написати це може бути таким:

```
<a onClick={
   () => {
     window.location.href = 'details.html';
   }
}
Перейти до деталей
</a>
```

Ми зробимо щось подібне до останнього, але замість глобального window.location використаємо проп навігації, який буде передано нашим екранним компонентам.

Перехід на новий екран

```
import * as React from 'react';
import { Button, View, Text } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack';

function HomeScreen({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text>Home Screen</Text>
      <Button
        title="Go to Details"
        onPress={() => navigation.navigate('Details')}
      />
    </View>
  );
}

// ... other code from the previous section
```

Давайте розберемо його:

navigation – проп навігації передається кожному екранному компоненту (визначенню) у навігаторі власного стеку (більше про це пізніше у розділі «Проп навігації в деталях»);

navigate('Details') – ми викликаємо функцію навігації (на пропорі навігації – імена складні!) з назвою маршруту, на який ми хочемо перемістити користувача.

Якщо ми викличемо navigation.navigate з назвою маршруту, яку ми не визначили в навігаторі, вона видасть помилку в розробці, а в продакшн-збірці нічого не станеться. Інакше кажучи, ми можемо переміщатися лише за маршрутами, визначеними у нашому навігаторі – ми не можемо переміщатися до довільного компонента.

Отже, тепер у нас є стек з двома маршрутами: 1) маршрут «Додому» 2) маршрут «Деталі». Що станеться, якщо ми знову перейдемо до маршруту Деталі з екрана Деталі?

Перехід до маршруту кілька разів

```
function DetailsScreen({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center'
}}>
      <Text>Details Screen</Text>
      <Button
        title="Go to Details... again"
        onPress={() => navigation.navigate('Details')}
      />
    </View>
  );
}
```

Якщо ви запустите цей код, то помітите, що при натисканні кнопки "Go to Details... again" нічого не відбувається! Це тому, що ми вже знаходимося на маршруті Деталі. Функція навігації приблизно означає «перейти до цього екрана», і якщо ви вже на цьому екрані, то цілком логічно, що вона нічого не зробить.

Припустімо, що ми хочемо додати ще один екран подробиць. Це досить поширене явище у випадках, коли ви передаєте деякі унікальні дані для кожного маршруту (докладніше про це пізніше, коли ми поговоримо про параметри!). Для цього ми можемо змінити navigate на

push. Це дозволяє нам висловити намір додати ще один маршрут незалежно від існуючої історії навігації.

```
<Button
  title="Go to Details... again"
  onPress={() => navigation.push('Details')}
/>
```

Кожного разу, коли ви викликаєте push, ми додаємо новий маршрут до стеку навігації. Коли ви викликаєте команду navigate, вона спочатку намагається знайти існуючий маршрут з такою назвою, і лише потім виштовхує новий маршрут, якщо його ще немає у стеку.

Повернення назад

У заголовку, наданому навігатором стека, буде автоматично додано кнопку повернення, коли можна повернутися назад з активного екрана (якщо у стеку навігації лише один екран, немає нічого, до чого можна повернутися, і тому кнопка повернення відсутня).

Іноді вам може знадобитися можливість програмно викликати таку поведінку, і для цього ви можете скористатися функцією navigation.goBack();.

```
function DetailsScreen({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text>Details Screen</Text>
      <Button
        title="Go to Details... again"
        onPress={() => navigation.push('Details')}
      />
      <Button title="Go to Home" onPress={() => navigation.navigate('Home')} />
      <Button title="Go back" onPress={() => navigation.goBack()} />
    </View>
  );
}
```

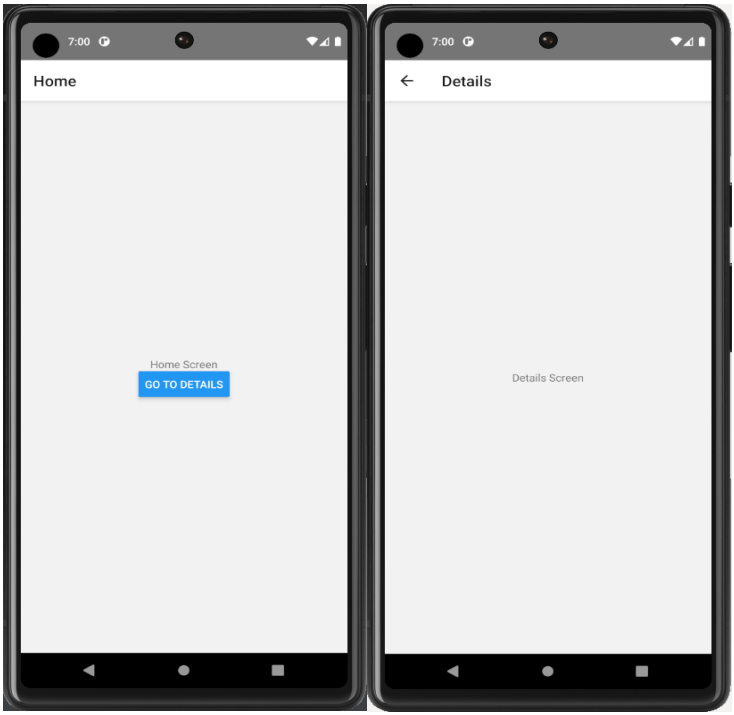


Рисунок 9.3 – Повернення назад

На Android React Navigation підключається до апаратної кнопки «Назад» і викликає функцію goBack(), коли користувач натискає її, таким чином поводячи себе так, як очікує користувач.

Іншою поширеною вимогою є можливість повернутися на кілька екранів назад – наприклад, якщо ви перебуваєте на кількох екранах у стеку і хочете закрити їх усі, щоб повернутися до першого екрана. У цьому випадку ми знаємо, що хочемо повернутися на Головний екран, тому можемо скористатися navigate('Home') (а не push! Спробуйте і побачите різницю). Іншою альтернативою може бути навігація.popToTop(), яка повертає до першого екрана у стеку.

```
function DetailsScreen({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text>Details Screen</Text>
```

```
        <Button
          title="Go to Details... again"
          onPress={() => navigation.push('Details')}
        />
        <Button title="Go to Home" onPress={() =>
navigation.navigate('Home')} />
        <Button title="Go back" onPress={() => navigation.goBack()} />
        <Button
          title="Go back to first screen in stack"
          onPress={() => navigation.popToTop()}
        />
      </View>
    );
  }
```

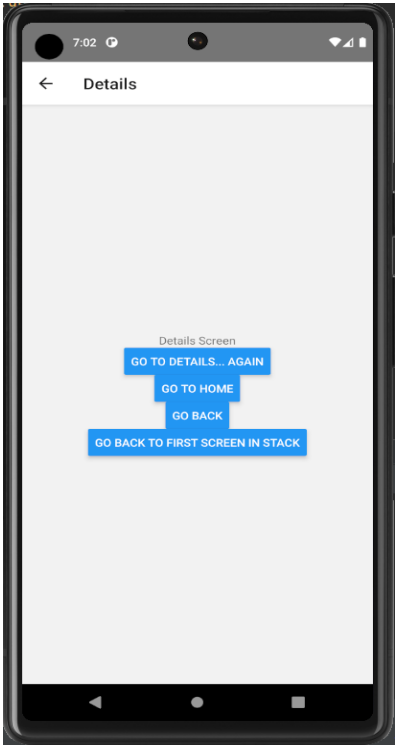


Рисунок 9.4 – Створені кнопки для навігації

Передача параметрів маршрутам

Тепер, коли ми знаємо, як створити навігатор стека з деякими маршрутами і здійснювати навігацію між ними, давайте подивимося, як ми можемо передавати дані маршрутам під час навігації на них.

Тут є дві складові:

1. Передати параметри маршруту, помістивши їх в об'єкт як другий параметр до функції navigation.navigate: navigation.navigate('RouteName', { /* параметри йдуть сюди */ })

2. Прочитати параметри у вашому екранному компоненті: route.params.

```
function HomeScreen({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center'
}}>
      <Text>Home Screen</Text>
      <Button
        title="Go to Details"
        onPress={() => {
          navigation.navigate('Details', {
            itemId: 86,
            otherParam: 'anything you want here',
          });
        }}
      />
    </View>
  );
}

function DetailsScreen({ route, navigation }) {
  const { itemId, otherParam } = route.params;
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center'
}}>
      <Text>Details Screen</Text>
      <Text>itemId: {JSON.stringify(itemId)}</Text>
      <Text>otherParam: {JSON.stringify(otherParam)}</Text>
      <Button
        title="Go to Details... again"
        onPress={() =>
          navigation.push('Details', {
            itemId: Math.floor(Math.random() * 100),
```

```
          })
        }
      />
      <Button title="Go to Home" onPress={() =>
navigation.navigate('Home')} />
      <Button title="Go back" onPress={() => navigation.goBack()} />
    </View>
  );
}
```

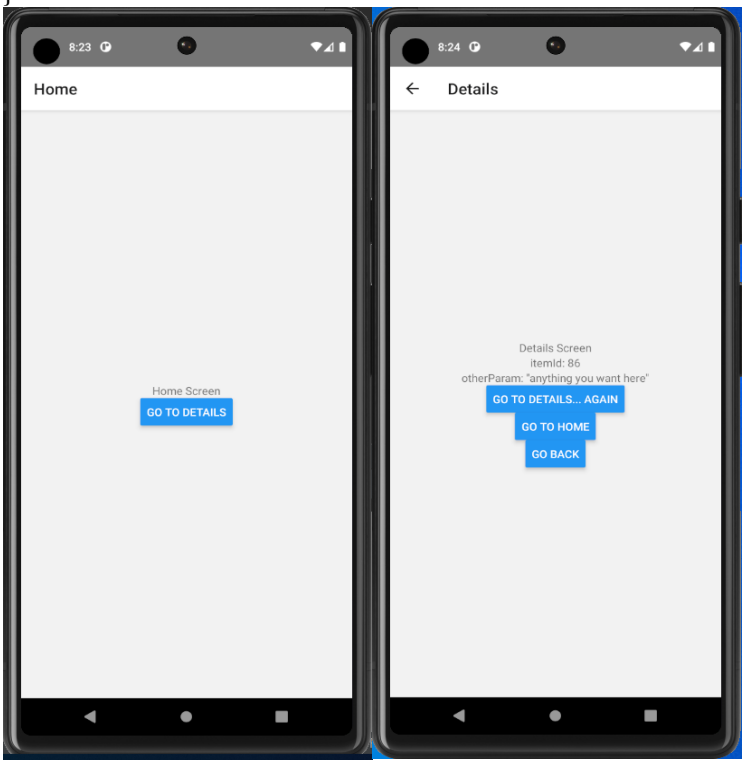


Рисунок 9.5 – Передані параметри

Початкові параметри

Ви можете передати деякі початкові параметри екрану. Якщо ви не вказали жодних параметрів під час переходу до цього екрана, буде використано початкові параметри. Вони також будуть неглибоко

об'єднані з будь-якими параметрами, які ви передасте. Початкові параметри можна вказати за допомогою пропса initialParams:

```
<Stack.Screen
  name="Details"
  component={DetailsScreen}
  initialParams={{ itemId: 42 }}
/>
```

Оновлення параметрів

Екрани також можуть оновлювати свої параметри, як і свій стан. Метод navigation.setParams дозволяє оновити параметри екрана. Зверніться до довідника API для setParams за більш детальною інформацією.

Базове використання:

```
navigation.setParams({
query: 'someText',
});
```

Передача параметрів на попередній екран

Параметри корисні не тільки для передачі деяких даних на новий екран, вони також можуть бути корисними для передачі даних на попередній екран. Наприклад, скажімо, у вас є екран із кнопкою створення допису, і кнопка створення допису відкриває новий екран для створення допису. Після створення допису ви хочете передати дані для нього назад на попередній екран.

Для цього ви можете використати метод navigate, який діє як goBack, якщо екран вже існує. Ви можете передати параметри методу navigate, щоб передати дані назад:

```
function HomeScreen({ navigation, route }) {
  React.useEffect(() => {
    if (route.params?.post) {
      // Post updated, do something with `route.params.post`
      // For example, send the post to the server
    }
  }, [route.params?.post]);

  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center'
}}>
      <Button
```

```
        title="Create post"
        onPress={() => navigation.navigate('CreatePost')}
      />
      <Text style={{ margin: 10 }}>Post: {route.params?.post}</Text>
    </View>
  );
}

function CreatePostScreen({ navigation, route }) {
  const [postText, setPostText] = React.useState('');

  return (
    <>
      <TextInput
        multiline
        placeholder="What's on your mind?"
        style={{ height: 200, padding: 10, backgroundColor: 'white' }}
        value={postText}
        onChangeText={setPostText}
      />
      <Button
        title="Done"
        onPress={() => {
          // Pass and merge params back to home screen
          navigation.navigate({
            name: 'Home',
            params: { post: postText },
            merge: true,
          });
        }}
      />
    </>
  );
}
```

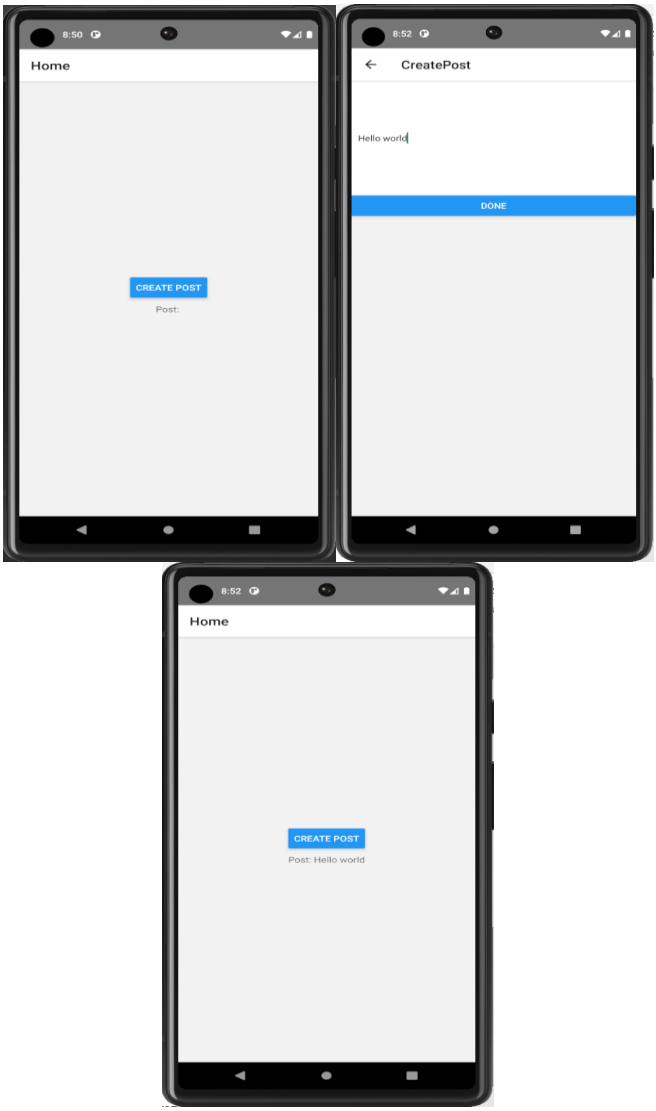


Рисунок 9.6 – Передача параметрів на попередній екран

Тут, після натискання кнопки «Готово», файл route.params на головному екрані буде оновлено, щоб відобразити текст повідомлення, яке ви пройшли в навігації.

Передача параметрів вкладеним навігаторам

Якщо у вас є вкладені навігатори, вам потрібно передати параметри трохи по-іншому. Наприклад, скажімо, у вас є навігатор всередині екрана Облікового запису, і ви хочете передати параметри на екран Налаштування всередині цього навігатора. Тоді ви можете передати параметри наступним чином:

```
navigation.navigate('Account', {
  screen: 'Settings',
  params: { user: 'jane' },
});
```

Навігація за допомогою панелей

Поширеним шаблоном навігації є використання панелі з лівого (іноді з правого) боку для переходу між екранами.

Перш ніж продовжити, спочатку встановіть і налаштуйте @react-navigation/drawer та його залежності.

Потім потрібно встановити і налаштувати бібліотеки, які потрібні навігатору ящиків:

Спочатку встановіть react-native-gesture-handler та react-native-reanimated.

```
import * as React from 'react';
import { Button, View } from 'react-native';
import { createDrawerNavigator } from '@react-navigation/drawer';
import { NavigationContainer } from '@react-navigation/native';

function HomeScreen({ navigation }) {
 return (
  <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
   <Button
    onPress={() => navigation.navigate('Notifications')}
    title="Go to notifications"
   />
  </View>
 );
}
function NotificationsScreen({ navigation }) {
 return (
  <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
   <Button onPress={() => navigation.goBack()} title="Go back home" />
  </View>
```

```
    );
  }

  const Drawer = createDrawerNavigator();

  export default function App() {
    return (
      <NavigationContainer>
        <Drawer.Navigator initialRouteName="Home">
          <Drawer.Screen name="Home" component={HomeScreen} />
          <Drawer.Screen                                    name="Notifications"
component={NotificationsScreen} />
        </Drawer.Navigator>
      </NavigationContainer>
    );
  }
```

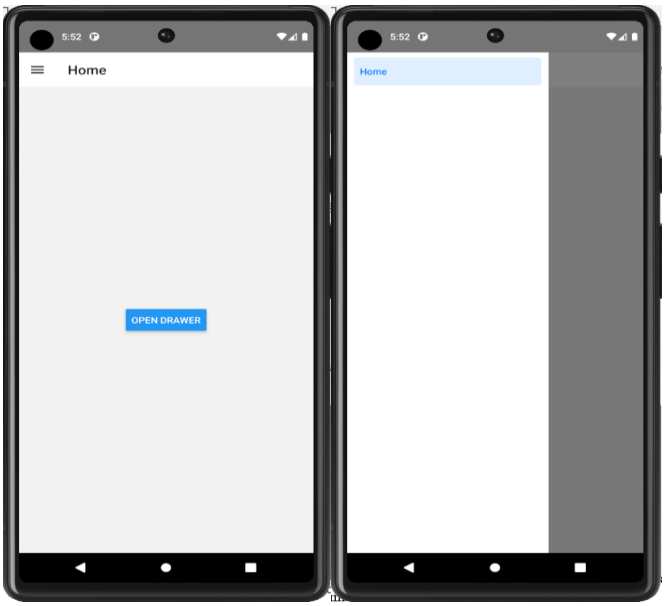


Рисунок 9.7 – Навігація за допомогою панелі

Кілька панелей

Іноді ми хочемо мати кілька шухляд на одному екрані: одну зліва і одну справа. Цього можна досягти, вклавши 2 навігатори шухляд.

Вкладення 2 навігаторів для панелей

Тут ми маємо 2 вкладені одна в одну шухляди, одна з яких розташована ліворуч, а інша праворуч:

```
import * as React from 'react';
import { Button, View } from 'react-native';
import { createDrawerNavigator } from '@react-navigation/drawer';
import { NavigationContainer } from '@react-navigation/native';

function HomeScreen({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Button onPress={() => navigation.openDrawer()} title="Open drawer" />
    </View>
  );
}

const LeftDrawer = createDrawerNavigator();

const LeftDrawerScreen = () => {
  return (
    <LeftDrawer.Navigator screenOptions={{ drawerPosition: 'left' }}>
      <LeftDrawer.Screen name="Home" component={HomeScreen} />
    </LeftDrawer.Navigator>
  );
};

const RightDrawer = createDrawerNavigator();

const RightDrawerScreen = () => {
  return (
    <RightDrawer.Navigator
      screenOptions={{ drawerPosition: 'right', headerShown: false }}
    >
      <RightDrawer.Screen name="HomeDrawer" component={LeftDrawerScreen} />
    </RightDrawer.Navigator>
  );
};

export default function App() {
  return (
    <NavigationContainer>
      <RightDrawerScreen />
```

```
      </NavigationContainer>
    );
  }
```

Але є одна проблема. Коли ми викликаємо navigation.openDrawer() на нашому HomeScreen, він завжди відкриває ліву шухляду, оскільки вона є безпосереднім батьком екрану.

Щоб вирішити цю проблему, нам потрібно використовувати навігацію.getParent для посилання на праву панель, яка є батьком лівої панелі. Таким чином, наш код буде виглядати так:

```
<Button onPress={() => navigation.openDrawer()} title="Open left drawer" />
<Button onPress={() => navigation.getParent().openDrawer()} title="Open right drawer" />
```

Однак це означає, що наша кнопка повинна знати про батьківські навігатори, що не є ідеальним варіантом. Якщо наша кнопка буде вкладена в інші навігатори, їй знадобиться декілька викликів getParent(). Щоб вирішити цю проблему, ми можемо використати проп id для визначення батьківського навігатора.

Щоб налаштувати вміст шухляди, ми можемо використати проп drawerContent для передачі у функцію, яка рендерить кастомний компонент.

Остаточний код виглядатиме так:

```
import * as React from 'react';
import { Button, Text, View } from 'react-native';
import { createDrawerNavigator } from '@react-navigation/drawer';
import { NavigationContainer } from '@react-navigation/native';

function HomeScreen({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Button
        onPress={() => navigation.getParent('LeftDrawer').openDrawer()}
        title="Open left drawer"
      />
      <Button
        onPress={() => navigation.getParent('RightDrawer').openDrawer()}
        title="Open right drawer"
      />
    </View>
  );
}
```

```
    function RightDrawerContent() {
      return (
        <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center'
}}>
          <Text>This is the right drawer</Text>
        </View>
      );
    }

    const LeftDrawer = createDrawerNavigator();

    function LeftDrawerScreen() {
      return (
        <LeftDrawer.Navigator
          id="LeftDrawer"
          screenOptions={{ drawerPosition: 'left' }}>
          <LeftDrawer.Screen name="Home" component={HomeScreen} />
        </LeftDrawer.Navigator>
      );
    }

    const RightDrawer = createDrawerNavigator();

    function RightDrawerScreen() {
      return (
        <RightDrawer.Navigator
          id="RightDrawer"
          drawerContent={(props) => <RightDrawerContent {...props} />}
          screenOptions={{
            drawerPosition: 'right',
            headerShown: false,
          }}>
          <RightDrawer.Screen                    name="HomeDrawer"
component={LeftDrawerScreen} />
        </RightDrawer.Navigator>
      );
    }

    export default function App() {
      return (
        <NavigationContainer>
          <RightDrawerScreen />
        </NavigationContainer>
      );
    }
```

Рисунок 9.8 – Використання декількох панелей

Завдання

Створіть мобільний застосунок із 3 екранами та навігацією по ним. Додайте основні компоненти на екрани і створіть книгу контактів, галерею або будь-який інший простий додаток.

Контрольні питання

1. Як влаштована нативна навігація на Android?
2. Що таке навігаційний стек?
3. Як передати параметри між роутами?

ЛАБОРАТОРНА РОБОТА №10 «ЗБЕРЕЖЕННЯ ДАНИХ У МОБІЛЬНОМУ ЗАСТОСУНКУ REACT NATIVE ЗА ДОПОМОГОЮ REDUX»

Теоретичні відомості

Redux – це контейнер із передбачуваним станом для застосунків JavaScript. Він допомагає вам застосунки, які поводяться узгоджено, працюють у різних середовищах (клієнтських, серверних і власних) і легко тестуються. На додачу до всього, він надає чудові можливості для розробників, такі як редагування коду в реальному часі в поєднанні з відладчиком, що дає змогу переглядати виконання коду, зупиняючись і перезапускаючи лише окремі його частини.

Store – це об'єкт Redux, який:

- містить state застосунку;
- відображає state через getState();
- може оновлювати стан, за допомогою dispatch();
- дозволяє реєструватися як слухач зміни стану при використанні методу subscribe().

Reducer – чиста функція, яка звертається до store і передає наступний стан дерева на підставі його попереднього стану, і застосовуваної дії (action). Чиста функція працює незалежно від стану програми і видає вихідне значення, приймаючи вхідне і не змінюючи нічого в ньому, і в решті програми.

Action – об'єкт, який лаконічно описує суть зміни, має в наявності слово type (як правило, то рядок).

Нижче наведено повний програмний код застосунку, детально розглянутого в лекції №7.

Файл App.js

```
import React from 'react'

import Books from './src/Books'
import rootReducer from './reducers/index'

import { Provider } from 'react-redux'
import { createStore } from 'redux'

const store = createStore(rootReducer)

export default class App extends React.Component {
  render() {
```

```
        return (
          <Provider store={store} >
            <Books />
          </Provider>
        )
      }
    }
```

Файл reducers/index.js

```
import { combineReducers } from 'redux'
import bookReducer from './bookReducer'

const rootReducer = combineReducers({
  bookReducer
})

export default rootReducer
```

Файл reducers/bookReducer.js

```
import uuid from 'react-native-uuid'
import { ADD_BOOK, REMOVE_BOOK } from '../src/actions'

const initialState = {
  books: [{ name: 'East of Eden', author: 'John Steinbeck', id: uuid.v4()
}]
}

const bookReducer = (state = initialState, action) => {
  switch(action.type) {
    case ADD_BOOK:
      return {
        books: [
         ...state.books,
         action.book
        ]
      }
    case REMOVE_BOOK:
      const index = state.books.findIndex(
          book => book.id === action.book.id)
       return {
          books: [
```

```
            ...state.books.slice(0, index),
            ...state.books.slice(index + 1)
          ]
        }
    default:
      return state
  }
}

export default bookReducer
```

Файл src/actions.js

```
export const ADD_BOOK = 'ADD_BOOK'
export const REMOVE_BOOK = 'REMOVE_BOOK'
import uuid from 'react-native-uuid'

export function addBook (book) {
  return {
    type: ADD_BOOK,
    book: {
      ...book,
      id: uuid.v4()
    }
  }
}

export function removeBook (book) {
  return {
    type: REMOVE_BOOK,
    book
  }
}
```

Файл src/Books.js

```
import React from 'react'
import {
  Text,
  View,
  ScrollView,
  StyleSheet,
  TextInput,
```

```
    TouchableOpacity
} from 'react-native'
import { addBook, removeBook } from './actions'

import { connect } from 'react-redux'

const initialState = {
  name: '',
  author: ''
}


class Books extends React.Component {
    state = initialState

    updateInput = (key, value) => {
    this.setState({
        ...this.state,
        [key]: value
    })
    }

    addBook = () => {
    this.props.dispatchAddBook(this.state)
        this.setState(initialState)
    }

    removeBook = (book) => {
        this.props.dispatchRemoveBook(book)
    }

    render() {
        const { books } = this.props
        return (
        <View style={styles.container}>
            <Text style={styles.title}>Books</Text>
            <ScrollView
                keyboardShouldPersistTaps='always'
                style={styles.booksContainer}
            >
                {
```

```
            books.map((book, index) => (
            <View style={styles.book} key={index}>
              <Text style={styles.name}>{book.name}</Text>
              <Text style={styles.author}>{book.author}</Text>
              <Text onPress={() => this.removeBook(book)}>
                Remove
              </Text>
            </View>
            ))
          }
        </ScrollView>
        <View style={styles.inputContainer}>
          <View style={styles.inputWrapper}>
            <TextInput
            value={this.state.name}
            onChangeText={value => this.updateInput('name', value)}
            style={styles.input}
            placeholder='Book name'
            />
            <TextInput
            value={this.state.author}
            onChangeText={value => this.updateInput('author', value)}
            style={styles.input}
            placeholder='Author Name'
            />
          </View>
          <TouchableOpacity onPress={this.addBook}>
            <View style={styles.addButtonContainer}>
              <Text style={styles.addButton}>+</Text>
            </View>
          </TouchableOpacity>
        </View>
      </View>
    )
    }
}
```

```
const styles = StyleSheet.create({
    inputContainer: {
        padding: 10,
        backgroundColor: '#ffffff',
        borderTopColor: '#ededed',
        borderTopWidth: 1,
        flexDirection: 'row',
        height: 100
    },
    inputWrapper: {
        flex: 1
    },
    input: {
        height: 44,
        padding: 7,
        backgroundColor: '#ededed',
        borderColor: '#ddd',
        borderWidth: 1,
        borderRadius: 10,
        flex: 1,
        marginBottom: 5
    },
    addButton: {
        fontSize: 28,
        lineHeight: 28
    },
    addButtonContainer: {
        width: 80,
        height: 80,
        backgroundColor: '#ededed',
        marginLeft: 10,
        justifyContent: 'center',
        alignItems: 'center',
        borderRadius: 20
    },
    container: {
        flex: 1
    },
```

```
    booksContainer: {
      borderTopWidth: 1,
      borderTopColor: '#ddd',
      flex: 1
    },
    title: {
      paddingTop: 30,
      paddingBottom: 20,
      fontSize: 20,
      textAlign: 'center'
    },
    book: {
      padding: 20
    },
    name: {
      fontSize: 18
    },
    author: {
      fontSize: 14,
      color: '#999'
    }
})

const mapStateToProps = (state) => ({
  books: state.bookReducer.books
})

const mapDispatchToProps = {
    dispatchAddBook: (book) => addBook(book),
    dispatchRemoveBook: (book) => removeBook(book)
}

export default connect(mapStateToProps, mapDispatchToProps)(Books)
```

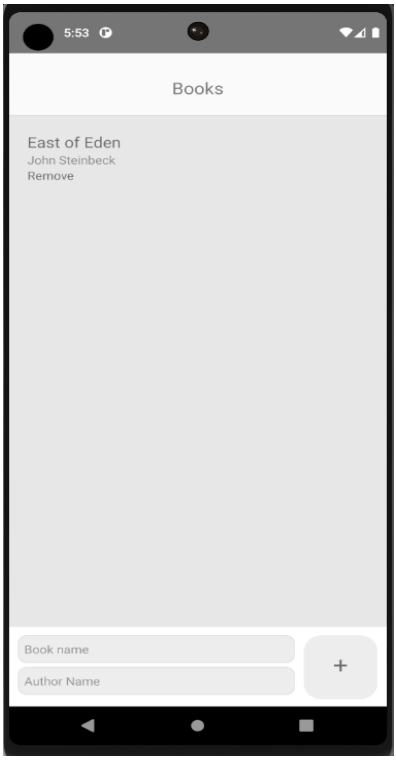


Рисунок 10.1 – Результат створення застосунку

Завдання

Ініціалізуйте Redux у мобільному застосунку. Створіть кошик для покупок криптовалют з використанням Redux. Має містити мінімум 8 найменувань товарів.

Контрольні питання

1. У яких випадках потрібен Redux?
2. Яким способом створюється сховище у Redux?
3. Що таке reducer?
4. Яку властивість обов’язково має містити action?

ЛАБОРАТОРНА РОБОТА №11 «РОБОТА З БД SQLITE У МОБІЛЬНИХ ЗАСТОСУНКАХ REACT NATIVE»

Теоретичні відомості

У практичні роботі ми розглянемо використання SQLite для локального збереження даних у наших React Native та Expo застосунках. SQLite доступний майже на всіх мобільних пристроях. Для доступу до нього потрібно писати SQL-запити, але дані повертаються у вигляді масивів та об'єктів javascript. Ми виконаємо CRUD-операції, щоб продемонструвати, як виконувати запити та оновлювати стани React.

Ми будемо використовувати пакет expo-sqlite:

expo install expo-sqlite

Тепер ми можемо імпортувати пакет і створити з'єднання з базою даних. Ми відкриваємо метод openDatabase, щоб створити з'єднання з базою даних. Якщо база даних не існує, то створюється нова.

```
import React from 'react';
import { View, Text, TouchableOpacity, ScrollView } from 'react-native';

import * as SQLite from 'expo-sqlite'
const db = SQLite.openDatabase('UserDatabase.db')
```

Операції

Далі розглянемо, як можна виконувати CRUD-операції над SQLite та оновлювати стан React. Ми будемо використовувати транзакції для взаємодії з базою даних за допомогою методу Database.transaction.

Database.transaction(callback, errorCallback, successCallback)

У кожному головному колбеку транзакції передається об'єкт Transaction, який можна використовувати для виконання запитів за допомогою методу Transaction.executeSql().

Transaction.executeSql(sqlStatementString, argumentsArray, successCallback, errorCallback)

SuccessCallback повертає ResultSets, який містить кількість зачеплених рядків rowsAffected, ідентифікатори вставки у випадку вставки та рядки даних у випадку зчитування.

Читання

Перший крок – це читання даних із таблиці кожного разу, коли додаток запускається. У методі транзакції ми використовуємо лише головний колбек, куди передається об'єкт транзакції tx. Ми використовуємо його для запуску стандартного SQL-запиту на читання.

```
db.transaction((tx) => {
    tx.executeSql(
      'SELECT * FROM table_user',
      [],
      (tx, results) => {
        var temp = [];
        for (let i = 0; i < results.rows.length; ++i)
          temp.push(results.rows.item(i));
        setFlatListItems(temp);
      }
    );
  });
```

Створення

Ми використовуємо той самий шаблон транзакції та executeSql, що й раніше, щоб вставити новий рядок у таблицю.

```
db.transaction(function (tx) {
    tx.executeSql(
      'INSERT INTO table_user (user_name, user_contact, user_address)
VALUES (?,?,?)',
      [userName, userContact, userAddress],
      (tx, results) => {
        console.log('Results', results.rowsAffected);
        if (results.rowsAffected > 0) {
          Alert.alert(
            'Success',
            'You are Registered Successfully',
            [
              {
                text: 'Ok',
                onPress: () => navigation.navigate('HomeScreen'),
              },
            ],
            { cancelable: false }
          );
        } else alert('Registration Failed');
```

```
        }
       );
      });
```

Оновлення

```
  db.transaction((tx) => {
      tx.executeSql(
       'SELECT * FROM table_user where user_id = ?',
       [inputUserId],
       (tx, results) => {
        var len = results.rows.length;
        if (len > 0) {
         let res = results.rows.item(0);
         updateAllStates(
          res.user_name,
          res.user_contact,
          res.user_address
         );
        } else {
         alert('No user found');
         updateAllStates('', '', '');
        }
       }
      );
     });

  …

     db.transaction((tx) => {
      tx.executeSql(
       'UPDATE table_user set user_name=?, user_contact=? ,
user_address=? where user_id=?',
       [userName, userContact, userAddress, inputUserId],
       (tx, results) => {
        console.log('Results', results.rowsAffected);
        if (results.rowsAffected > 0) {
         Alert.alert(
          'Success',
          'User updated successfully',
          [
           {
```

```
              text: 'Ok',
              onPress: () => navigation.navigate('HomeScreen'),
            },
          ],
          { cancelable: false }
        );
      } else alert('Updation Failed');
    }
  );
 });
};
```

Видалення

```
db.transaction((tx) => {
    tx.executeSql(
      'DELETE FROM  table_user where user_id=?',
      [inputUserId],
      (tx, results) => {
        console.log('Results', results.rowsAffected);
        if (results.rowsAffected > 0) {
          Alert.alert(
            'Success',
            'User deleted successfully',
            [
              {
                text: 'Ok',
                onPress: () => navigation.navigate('HomeScreen'),
              },
            ],
            { cancelable: false }
          );
        } else {
          alert('Please insert a valid User Id');
        }
      }
    );
  });
```

Повний програмний код мобільного застосунку для роботи з базою даних SQLite можна подивитись у лекції №8 (в тім застосунок створений за допомогою React Native CLI, а не Expo CLI)..

Завдання

Створіть мобільний застосунок – інтернет-магазин зі списком товарів із БД.

Контрольні питання

1. Які інші способи роботи БД у React Native ви знаєте?
2. Який компонент і бібліотека дають змогу організувати локальне сховище в React Native?
3. Що визначають параметри методу executeSQL?

ЛАБОРАТОРНА РОБОТА №12
«РОБОТА З БД REALM У МОБІЛЬНИХ ЗАСТОСУНКАХ REACT NATIVE»

Теоретичні відомості

Realm – це система управління об'єктними базами даних з відкритим вихідним кодом, спочатку для мобільних пристроїв, також доступна для таких платформ, як Xamarin або React Native, та інших, включаючи десктопні додатки, і ліцензована за ліцензією Apache License. Ми можемо виконувати всі CRUD-транзакції в цій базі даних і набагато швидше, ніж в базі даних SQLite. Давайте почнемо з Realm.

У цьому прикладі ми створимо HomeScreen з можливістю переходу на інші екрани, такі як:

- **RegisterUser:** Реєстрація користувача. (Створити/Вставити)
- **ViewAllUser:** для перегляду всіх користувачів. (Читати)
- **ViewUser:** Переглянути окремих користувачів за ідентифікатором. (Читати)
- **UpdateUser:** оновити користувача. (Оновити)
- **DeleteUser:** видалити користувача.

Ми матимемо деякі власні компоненти, такі як Mybutton, Mytext, Mytextinput, які використовуватимуться замість реактивних Button, Text і TextInput.

Для використання Realm потрібно імпортувати бібліотеку ось так:

```
import { Realm } from 'realm';
```

і відкрити базу даних за допомогою

```
let realm = new Realm({ path: 'UserDatabase.realm' })
```

Тепер, коли вам потрібно зробити якийсь виклик до бази даних, ви можете використовувати realm для виконання запиту до бази даних

```
realm.write(() => {
```

Тут виконується операція CRUD (операція читання може бути виконана безпосередньо)

```
});
```

Якщо ви раніше встановили глобальний пакет react-native-cli, будь ласка, видаліть його, оскільки він може спричинити непередбачувані проблеми:

npm uninstall -g react-native-cli @react-native-community/cli

Запустіть наступні команди, щоб створити новий React Native проєкт

npx react-native init ProjectName

Щоб встановити залежності, відкрийте термінал і перейдіть до вашого проєкту:

cd Назва_проєкту

1. Встановіть залежність realm для використання RealM

npm install realm --save

2. Встановіть залежність react-navigation для імпорту createAppContainer

npm install react-navigation --save

3. Інші допоміжні бібліотеки для react-navigation

npm install react-native-gesture-handler react-native-safe-area-context @react-native-community/masked-view react-native-screens react-native-reanimated --save

4. Встановіть react-navigation-stack для імпорту createStackNavigator

npm install react-navigation-stack --save

Ці команди скопіюють усі залежності до вашого каталогу node_module.

Структура проєкту

У цьому прикладі ми виконаємо повну операцію CRUD. Будь ласка, створіть наступну структуру проєкту і скопіюйте код, наведений нижче. (рис. 12.1)

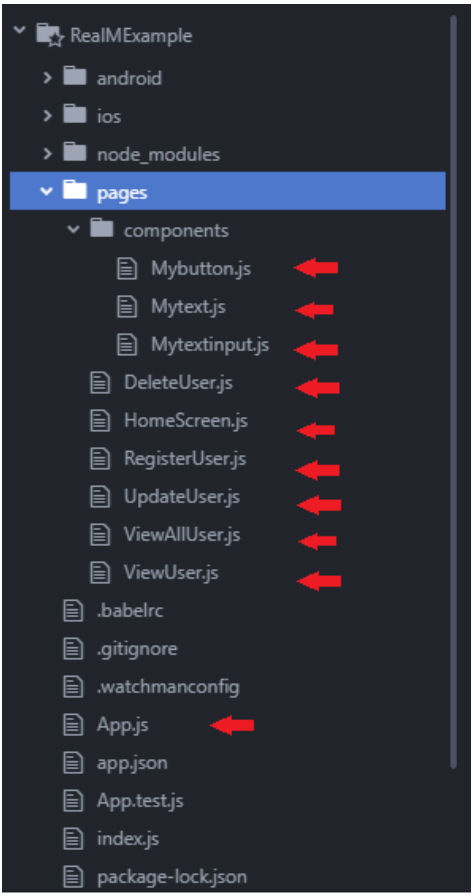


Рисунок 12.1 – Структура проєкту

Повний код до прикладу мобільного застосунку до поточної лабораторної роботи наведено в окремому файлі, що розташований поруч.

Додавання (реєстрація) користувача

```
register_user = () => {
  var that = this;
  const { user_name } = this.state;
  const { user_contact } = this.state;
```

```
  const { user_address } = this.state;
  if (user_name) {
    if (user_contact) {
      if (user_address) {
        realm.write(() => {
          var ID =
            realm.objects('user_details').sorted('user_id', true).length > 0
              ? realm.objects('user_details').sorted('user_id', true)[0]
                  .user_id + 1
              : 1;
          realm.create('user_details', {
            user_id: ID,
            user_name: that.state.user_name,
            user_contact: that.state.user_contact,
            user_address: that.state.user_address,
          });
          Alert.alert(
            'Success',
            'You are registered successfully',
            [
              {
               text: 'Ok',
               onPress: () => that.props.navigation.navigate('HomeScreen'),
              },
            ],
            { cancelable: false }
          );
        });
      } else {
        alert('Please fill Address');
      }
    } else {
      alert('Please fill Contact Number');
    }
  } else {
    alert('Please fill Name');
  }
};
```

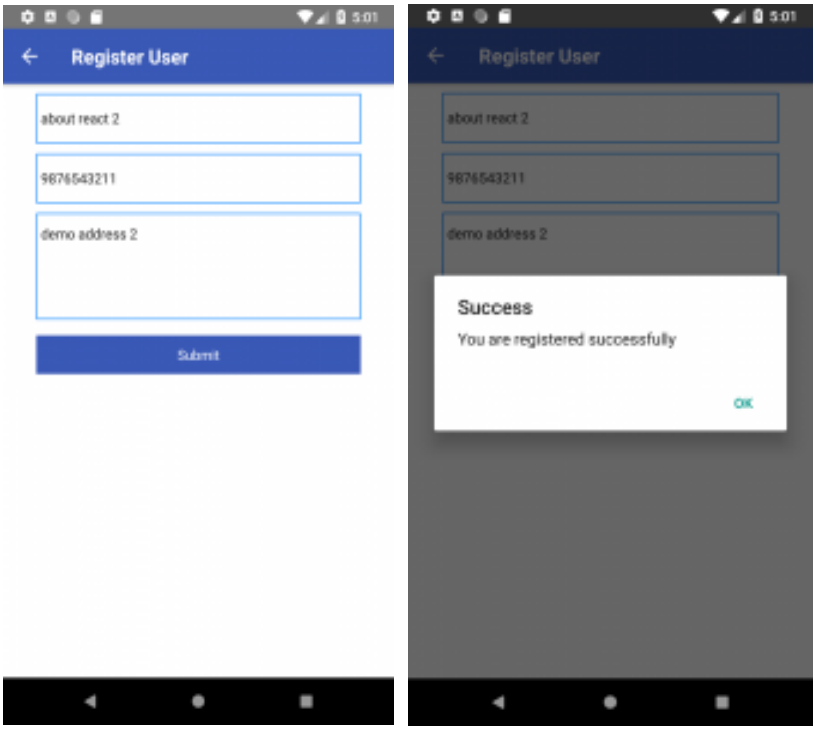


Рисунок 12.2 – Реєстрація користувача

Оновлення користувача

Метод searchUser також використовується і при відображенні певного користувача.

```
searchUser = () => {
  const { input_user_id } = this.state;
  console.log(this.state.input_user_id);
  var user_details = realm
    .objects('user_details')
    .filtered('user_id =' + input_user_id);
  console.log(user_details);
  if (user_details.length > 0) {
    this.setState({
      user_name: user_details[0].user_name,
    });
```

```
    this.setState({
      user_contact: user_details[0].user_contact,
    });
    this.setState({
      user_address: user_details[0].user_address,
    });
  } else {
    alert('No user found');
    this.setState({
      user_name: '',
    });
    this.setState({
      user_contact: '',
    });
    this.setState({
      user_address: '',
    });
  }
};

updateUser = () => {
  var that = this;
  const { input_user_id } = this.state;
  const { user_name } = this.state;
  const { user_contact } = this.state;
  const { user_address } = this.state;
  if (input_user_id) {
    if (user_name) {
      if (user_contact) {
        if (user_address) {
          realm.write(() => {
            var ID = this.state.input_user_id;
            console.log('ID', ID);
            var obj = realm
              .objects('user_details')
              .filtered('user_id =' + this.state.input_user_id);
            console.log('obj', obj);
            if (obj.length > 0) {
              obj[0].user_name = this.state.user_name;
```

```
            obj[0].user_contact = this.state.user_contact;
            obj[0].user_address = this.state.user_address;
            Alert.alert(
              'Success',
              'User updated successfully',
              [
                {
                  text: 'Ok',
                  onPress: () =>
                    that.props.navigation.navigate('HomeScreen'),
                },
              ],
              { cancelable: false }
            );
          } else {
            alert('User Updation Failed');
          }
        });
      } else {
        alert('Please fill Address');
      }
    } else {
      alert('Please fill Contact Number');
    }
  } else {
    alert('Please fill Name');
  }
 } else {
  alert('Please fill User Id');
 }
};
```

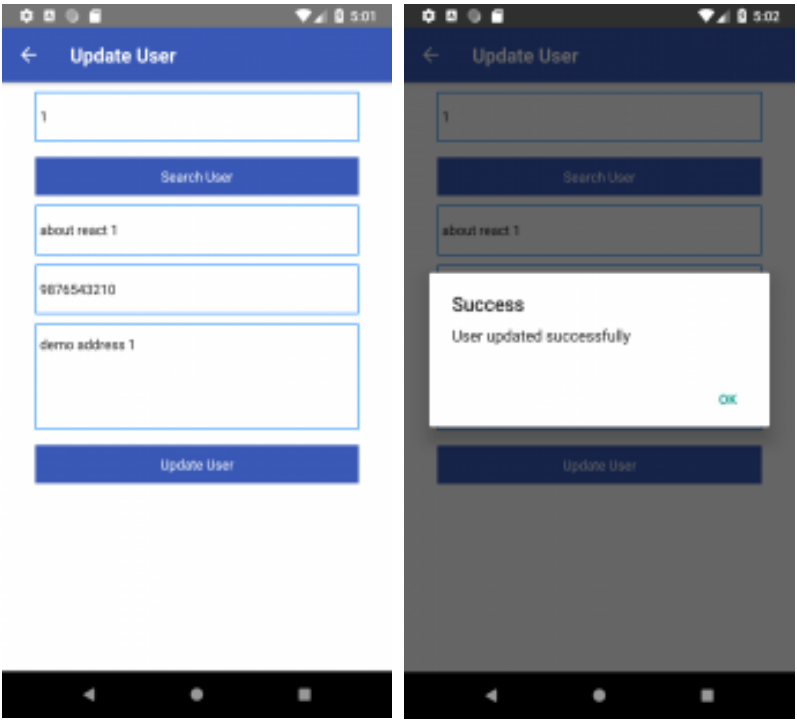


Рисунок 12.3 – Оновлення користувача

```
deleteUser = () => {
  var that = this;
  const { input_user_id } = this.state;
  realm.write(() => {
    var ID = this.state.input_user_id;
    if (
      realm.objects('user_details').filtered('user_id =' + input_user_id)
        .length > 0
    ) {
      realm.delete(
        realm.objects('user_details').filtered('user_id =' + input_user_id)
      );
      var user_details = realm.objects('user_details');
      console.log(user_details);
      Alert.alert(
```

```
          'Success',
          'User deleted successfully',
          [
            {
              text: 'Ok',
              onPress: () => that.props.navigation.navigate('HomeScreen'),
            },
          ],
          { cancelable: false }
        );
      } else {
        alert('Please insert a valid User Id');
      }
    });
  };
```

Рисунок 12.4 – Видалення користувача

Завдання

Переробіть мобільний застосунок із минулої лабораторної роботи з використанням мобільної СКБД Realm.

Контрольні питання

1. Як підключити мобільну СКБД Realm до проєкту React Native?
2. Що представляє собою схема бази даних Realm?
3. Як додати властивість сутоності Realm?
4. Які типи відношень існують між сутностями в Realm? Як реалізувати їх програмно

ЛАБОРАТОРНА РОБОТА №13
«РОБОТА З REST API У REACT NATIVE»

Теоретичні відомості

REST API (Representational State Transfer) є стандартом для обміну даними між клієнтом (мобільним застосунком) та сервером. У React Native для роботи з REST API зазвичай використовується вбудована функція fetch або бібліотека axios. Ці інструменти дозволяють виконувати HTTP-запити (GET, POST, PUT, DELETE) для отримання або надсилання даних.

Встановлення axios

Для спрощення роботи з HTTP-запитами встановіть бібліотеку axios:

```
npm install axios
```

Основні операції з REST API:

- GET: Отримує дані з сервера.
- POST: Надсилає дані на сервер для створення ресурсу.
- PUT: Оновлює існуючий ресурс.
- DELETE: Видаляє ресурс.

Приклад роботи з REST API

У цьому прикладі ми створимо застосунок, який отримує список користувачів із публічного API (JSONPlaceholder) і відображає їх у FlatList. Користувач може додати нового користувача через POST-запит. (рис. 13.1)

```
import React, { useState, useEffect } from 'react';
import { View, Text, TextInput, Button, FlatList, StyleSheet, Alert } from 'react-native';
import axios from 'axios';

export default function App() {
  const [users, setUsers] = useState([]);
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');

  useEffect(() => {
```

```
    fetchUsers();
  }, []);

  const fetchUsers = async () => {
    try {
      const response = await
axios.get('https://jsonplaceholder.typicode.com/users');
      setUsers(response.data);
    } catch (error) {
      Alert.alert('Error', 'Failed to fetch users');
    }
  };

  const addUser = async () => {
    if (!name || !email) {
      Alert.alert('Error', 'Please fill in all fields');
      return;
    }
    try {
      const response = await
axios.post('https://jsonplaceholder.typicode.com/users', {
        name,
        email,
      });
      setUsers([...users, response.data]);
      setName('');
      setEmail('');
      Alert.alert('Success', 'User added');
    } catch (error) {
      Alert.alert('Error', 'Failed to add user');
    }
  };

  const renderItem = ({ item }) => (
    <View style={styles.userItem}>
      <Text>{item.name}</Text>
      <Text>{item.email}</Text>
    </View>
  );

  return (
```

```
    <View style={styles.container}>
      <Text style={styles.title}>Users List</Text>
      <TextInput
        style={styles.input}
        placeholder="Name"
        value={name}
        onChangeText={setName}
      />
      <TextInput
        style={styles.input}
        placeholder="Email"
        value={email}
        onChangeText={setEmail}
      />
      <Button title="Add User" onPress={addUser} />
      <FlatList
        data={users}
        renderItem={renderItem}
        keyExtractor={(item) => item.id.toString()}
      />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20,
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 20,
  },
  input: {
    height: 40,
    borderColor: '#ccc',
    borderWidth: 1,
    marginBottom: 10,
    paddingHorizontal: 10,
  },
```

```
  userItem: {
    padding: 10,
    borderBottomWidth: 1,
    borderBottomColor: '#ccc',
  },
});
```

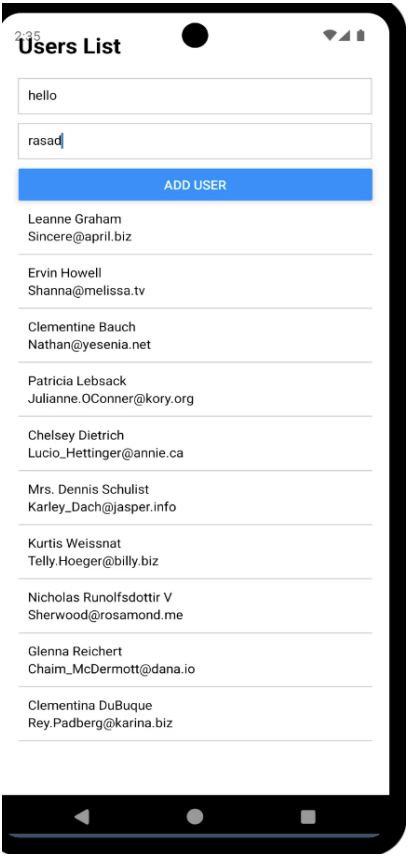


Рисунок 13.1 – Приклад роботи з REST API

Обробка помилок

При роботі з API важливо обробляти помилки, такі як відсутність мережі або некоректна відповідь сервера. Використовуйте try/catch із Alert для інформування користувача.

Оптимізація

Кешування: Зберігайте отримані дані в AsyncStorage для зменшення кількості запитів.

Пагінація: Для великих списків використовуйте параметри page або limit у запитах.

Завдання

Створіть мобільний застосунок, який отримує список постів із JSONPlaceholder API та відображає їх у FlatList. Додайте можливість створювати новий пост через POST-запит.

Контрольні питання

1. Що таке REST API і які типи HTTP-запитів він підтримує?
2. Яка різниця між fetch і axios для роботи з API?
3. Як обробляти помилки при виконанні HTTP-запитів?
4. Як реалізувати кешування даних із API?

ЛАБОРАТОРНА РОБОТА №14
«РОБОТА З ГЕОЛОКАЦІЄЮ У REACT NATIVE»

Теоретичні відомості

React Native дозволяє отримувати геолокаційні дані пристрою за допомогою бібліотеки expo-location. Ця бібліотека надає доступ до координат (широта, довгота), інформації про швидкість, висоту тощо. Для роботи з геолокацією потрібні дозволи користувача.

Встановлення expo-location

Встановіть бібліотеку:
npx expo install expo-location

Дозволи

Для доступу до геолокації потрібно запитати дозволи за допомогою requestForegroundPermissionsAsync. На iOS також потрібно додати ключі до app.json:

```
{
  "expo": {
    "ios": {
      "infoPlist": {
        "NSLocationWhenInUseUsageDescription": "This app needs access to your location to show your position."
      }
    }
  }
}
```

Приклад роботи з геолокацією

У цьому прикладі ми створимо застосунок, який отримує поточні координати користувача та відображає їх на екрані. (рис. 14.2) Перед цим застосунок запитає дозвіл на геолокацію (рис. 14.1)

```
import React, { useState, useEffect } from 'react';
import { View, Text, Button, StyleSheet, Alert } from 'react-native';
import * as Location from 'expo-location';

export default function App() {
```

```
  const [location, setLocation] = useState(null);
  const [errorMsg, setErrorMsg] = useState(null);

  useEffect(() => {
    (async () => {
      let { status } = await Location.requestForegroundPermissionsAsync();
      if (status !== 'granted') {
        setErrorMsg('Permission to access location was denied');
        return;
      }

      let location = await Location.getCurrentPositionAsync({});
      setLocation(location);
    })();
  }, []);

  const getLocation = async () => {
    try {
      let location = await Location.getCurrentPositionAsync({});
      setLocation(location);
      setErrorMsg(null);
    } catch (error) {
      setErrorMsg('Failed to fetch location');
    }
  };

  let text = 'Waiting...';
  if (errorMsg) {
    text = errorMsg;
  } else if (location) {
    text = `Latitude: ${location.coords.latitude}, Longitude: ${location.coords.longitude}`;
  }

  return (
    <View style={styles.container}>
      <Text style={styles.title}>Geolocation Example</Text>
      <Text style={styles.text}>{text}</Text>
      <Button title="Refresh Location" onPress={getLocation} />
    </View>
  );
}
```

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    padding: 20,
  },
  title: {
    fontSize: 24,
    marginBottom: 20,
  },
  text: {
    fontSize: 18,
    marginBottom: 20,
  },
});
```

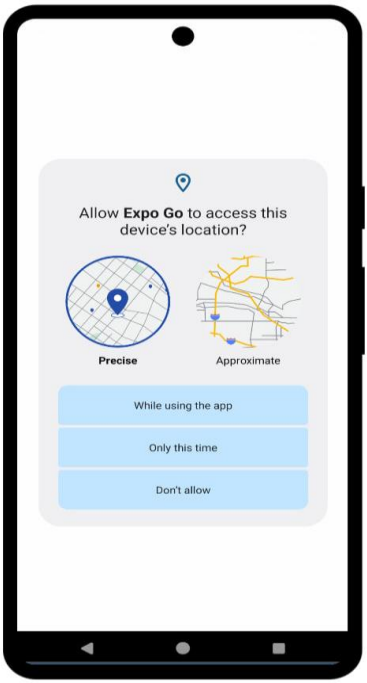


Рисунок 14.1 – Приклад запиту на роботу з геолокацією

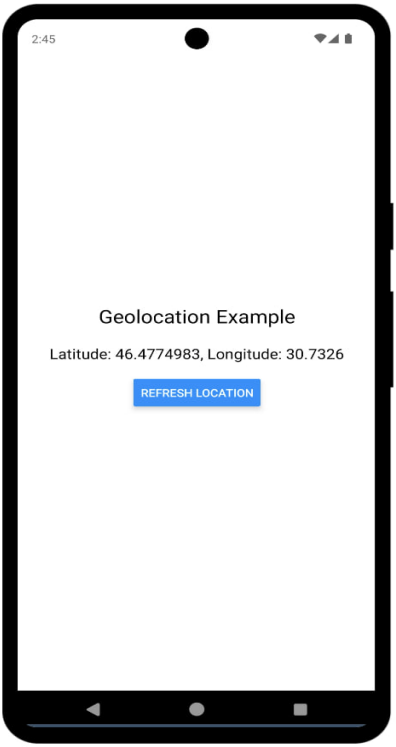


Рисунок 14.2 – Приклад роботи з геолокацією

Обмеження

– Точність: Точність геолокації залежить від пристрою та умов (наприклад, GPS чи Wi-Fi).

– Дозволи: Без дозволів користувача геолокація недоступна.

– Фоновий режим: Для фонової геолокації потрібні додаткові налаштування та дозволи.

.

Завдання

Створіть мобільний застосунок, який отримує поточну геолокацію користувача та зберігає історію координат у AsyncStorage. Додайте можливість відображення історії у FlatList.

Контрольні питання

1. Яка бібліотека використовується для роботи з геолокацією в React Native?
2. Як отримати дозволи на доступ до геолокації?
3. Які дані можна отримати з об’єкта геолокації?
4. Які обмеження має геолокація в мобільних застосунках? Як реалізувати кешування даних із API?

ВАРІАНТИ ЗАВДАНЬ ДЛЯ ЛАБОРАТОРНИХ РОБІТ 11-12

Здобувач може запропонувати свій варіант, що *суттєво* відрізняється від наведених нижче. Можливість вибору свого варіанту *узгоджується* з викладачем.

№	Завдання
1	Розробка інтернет-магазину харчових продуктів.
2	Розробка інтернет-магазину автомобілів.
3	Розробка інтернет-супермаркету.
4	Розробка інтернет-аптеки.
5	Розробка інтернет-магазину пекарні.
6	Розробка інтернет-магазину біжутерії.
7	Розробка інтернет-магазину аксесуарів для гаджетів.
8	Розробка інтернет-магазину гаджетів.
9	Розробка інтернет-магазину побутової техніки.
10	Розробка інтернет-магазину книг.
11	Розробка інтернет-магазину іграшок.
12	Розробка інтернет-магазину комп'ютерних ігор.
13	Розробка інтернет-магазину програм.
14	Розробка інтернет-магазину домашніх рослин.
15	Розробка інтернет-магазину сільськогосподарської техніки.
16	Розробка інтернет-магазину товарів для краси.
17	Розробка інтернет-магазину товарів для ремонту оселі.
18	Розробка інтернет-магазину спортивних товарів.
19	Розробка інтернет-магазину товарів для будинку.
20	Розробка інтернет-магазину меблів.
21	Розробка інтернет-магазину здорового харчування.
22	Розробка інтернет-магазину одягу та взуття.
23	Розробка інтернет-магазину для автомайстерень.
24	Розробка інтернет-магазину квитків на поїздки.
25	Розробка інтернет-магазину картин.
26	Розробка інтернет-магазину аудіо та відеопродукції.
27	Розробка інтернет-магазину музичних інструментів.
28	Розробка інтернет-магазину аудіо та відеотехніки.
29	Розробка інтернет-магазину товарів для кухні.
30	Розробка інтернет-магазину спортивного одягу та взуття.
31	Розробка інтернет-магазину екологічних товарів та продуктів.
32	Розробка інтернет-магазину мистецьких робіт та ручної роботи.
33	Розробка інтернет-магазину подарунків та сувенірів.
34	Розробка інтернет-магазину товарів для тварин та домашніх улюбленців.

№	Завдання
35	Розробка інтернет-магазину електроніки та гаджетів для геймерів.
36	Розробка інтернет-магазину товарів для активного відпочинку та подорожей.
37	Розробка інтернет-магазину товарів для фітнесу та здорового способу життя.
38	Розробка інтернет-магазину товарів для розвитку дітей.
39	Розробка інтернет-магазину товарів для гуртожитків та студентів.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. bin Uzayr S. Working with React Native. Mastering React Native. Boca Raton, 2022. P. 15–32. URL: https://doi.org/10.1201/9781003310440-2 (date of access: 12.04.2025).
2. Learning React: Modern Patterns for Developing React Apps. O'Reilly Media, 2020. 310 p.
3. Oates J. React Native. VTC, Incorporated, 2022.
4. He M. H. Foundations of React. Creating Apps with React Native. Berkeley, CA, 2022. P. 41–88. URL: https://doi.org/10.1007/978-1-4842-8042-3_2 (date of access: 12.04.2025).
5. Jakob Iversen & Michael Eierman. Mobile App Development for iOS and Android. Prospect Press, 2020. 140 p.
6. Derks R. React Projects: Build Advanced Cross-Platform Projects with React and React Native to Become a Professional Developer. Packt Publishing, Limited, 2021.
7. Adam Boduch, Roy Derks, Mikhail Sakhniuk. React and React Native: Build Cross-Platform JavaScript Applications with Native Power for the Web, Desktop, and Mobile. Packt Publishing, Limited, 2022. 606 p.
8. Masiello E., Friedmann J. Mastering React Native. Packt Publishing - ebooks Account, 2020. 496 p.
9. Rauter M., Wachtler J., Ebner M. Porting a Native Android App to iOS. International Journal of Interactive Mobile Technologies (iJIM). 2023. Vol. 17, no. 23. P. 20–31. URL: https://doi.org/10.3991/ijim.v17i23.43829 (date of access: 12.04.2025).
10. Narayn H. Just React!: Learn React the React Way. Apress L. P., 2022. 340 p.
11. Elrom E. Learn React Basic Concepts. React and Libraries. Berkeley, CA, 2021. P. 1–19. URL: https://doi.org/10.1007/978-1-4842-6696-0_1 (date of access: 12.04.2025).

Виробничо-практичне видання

Методична серія. Випуск 464

Гліб Валентинович Горбань,
Віктор Сергійович Раленко

ПРОГРАМУВАННЯ МОБІЛЬНИХ ПРИСТРОЇВ

Методичні рекомендації
до виконання лабораторних робіт

Редактор *О. Михайлова*
Комп'ютерна верстка, дизайн обкладинки *К. Гросу-Грабарчук*
Друк *С. Волинець*. Фальцювально-палітурні роботи *О. Мішалкіна.*

Підп. до друку 06.10.2025.
Формат 60x84 $^{1}/_{16}$. Папір офсет.
Гарнітура «Times New Roman». Друк різограф.
Ум. друк. арк. 8,8. Обл.-вид. арк. 3,4.
Тираж 50 пр. Зам. № 7029.

Видавець і виготовлювач: ЧНУ ім. Петра Могили.
54003, м. Миколаїв, вул. 68 Десантників, 10.
Тел.: 8 (0512) 50–03–32, 8 (0512) 76–55–81, e-mail: rector@chmnu.edu.ua.
Свідоцтво суб'єкта видавничої справи ДК № 6124 від 05.04.2018.