Розробка мобільних додатків

Лекція 4 - Навігація

Навігація

Навігація - одна з **ключових складових архітектури мобільного застосунку**.
Вона визначає не лише те, **як користувач переходить між екранами**, а й **як організований стан, логіка та потоки взаємодії в додатку**.

У React Native навігація тісно пов’язана з:

- управлінням станом екранів,
- життєвим циклом компонентів,
- UX-патернами мобільних платформ (Android / iOS),
- масштабованістю застосунку.

Неправильно спроєктована навігація призводить до:

- заплутаних користувацьких сценаріїв,
- дублювання стану,
- складної підтримки та розширення проєкту.

Web vs Mobile

У **web-застосунках** навігація базується на URL, який є джерелом істини для інтерфейсу. Перехід між сторінками змінює адресу, а браузер формує лінійну історію переходів. Під час навігації попередній екран фактично перемальовується або знищується, а кнопка «Назад» повертає користувача до попереднього URL.

У **React Native** навігація не прив’язана до URL і є частиною внутрішнього стану застосунку. Екрани організовані у стек навігації: новий екран додається поверх попереднього, який залишається змонтованим у пам’яті. Кнопка «Назад» виконує видалення верхнього екрана зі стеку, повертаючи користувача до попереднього стану інтерфейсу.

Ключова різниця полягає в тому, що у web навігація керує маршрутами, а в React Native - станом і життєвим циклом екранів.

Типологія Навігації: Базові патерни

1. Stack Navigation (Стек):

- **Логіка:** LIFO (Last In, First Out). Екрани накладаються шарами.
- **Кейс:** Деталізація контенту. *Список товарів → Товар → Оплата.*
- **Дія:** Push (глибше) / Pop (назад).

2. Bottom Tabs (Таби):

- **Логіка:** Паралельні незалежні стеки.
- **Кейс:** Головне меню додатку. *Instagram.*
- **Особливість:** Стан вкладок зберігається при перемиканні.

3. Drawer Navigation (Бокове меню):

- **Логіка:** Прихована панель навігації.
- **Кейс:** Другорядні розділи або складна ієрархія. *Gmail, Telegram.*

React Navigation

React Navigation - це де-факто стандартна бібліотека навігації для React Native застосунків. Вона реалізує навігацію як **керування станом екранів**, а не як маршрутизацію за URL. Бібліотека дозволяє будувати навігацію з окремих навігаційних контейнерів, які відповідають різним користувацьким сценаріям.

Гібридний підхід: поєднує гнучкість JavaScript та продуктивність нативних компонентів.

1. **@react-navigation/native (JS Layer):**
   - Відповідає за логіку маршрутизації.
   - Зберігає дерево навігації та параметри.
   - Повністю написана на JavaScript.
2. **react-native-screens (Native Layer):**
   - Використовує нативні оптимізації OS.
   - Дозволяє "відключати" рендер невидимих екранів для економії пам'яті.
   - Інтегрується з системними контролерами.

@react-navigation/native

7.1.28 • Public • Published 20 days ago

Readme | Code Beta | 5 Dependencies | 1572 Dependents | 313 Versions

@react-navigation/native

React Native integration for React Navigation.

Installation instructions and documentation can be found on the React Navigation website.

Keywords

react-native react-navigation ios android

Install

```
npm i @react-navigation/native
```

Repository

github.com/react-navigation/react-navigation

Homepage

reactnavigation.org

Weekly Downloads

2 898 368

Встановлення та Залежності

Для коректної роботи необхідний набір базових бібліотек.

1. Встановлення ядра: Це обов'язковий контейнер для керування станом навігації.

```
npx expo install @react-navigation/native
```

2. Встановлення нативних залежностей: Необхідні для коректної роботи жестів, анімацій та безпечних зон (safe areas).

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

react-native-safe-area-context

react-native-safe-area-context потрібна для коректної роботи інтерфейсу в межах **безпечної зони екрана** на різних мобільних пристроях.

У сучасних смартфонах екран не є прямокутним. Є системні області, в які не можна «залізати» контентом: вирізи (notch), статус-бар, нижня системна панель, індикатор жестів. Ці зони відрізняються між Android та iOS і навіть між моделями одного виробника.

`react-native-safe-area-context` визначає **реальні безпечні відступи** (top, bottom, left, right) для конкретного пристрою і передає їх у React Native застосунок. Завдяки цьому:

- контент не перекривається системними елементами;
- хедери, таби та кнопки розташовуються коректно на всіх екранах;
- навігація виглядає однаково правильно на iOS і Android.

У контексті **React Navigation** ця бібліотека критично важлива, бо навігаційні елементи (Stack Header, Bottom Tabs) автоматично підлаштовуються під safe area. Без неї заголовки можуть «заїжджати» під статус-бар, а нижні таби — під системну навігацію.

Встановлення: **Stack Navigator**

Після встановлення ядра React Navigation підключається конкретний тип навігатора.

Native Stack - це стекова навігація, реалізована через **нативні компоненти платформ** (Android / iOS), а не через JavaScript-реалізацію.

npx expo install @react-navigation/native-stack

Ініціалізація

Вхідна точка навігації `NavigationContainer` — це обгортка, яка відповідає за дерево навігації та історію переходів. Вона має бути **кореневим елементом** застосунку (зазвичай у `App.js`).

```
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack'

const Stack = createNativeStackNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator id='mainStack'>
        {/* Визначаємо екрани */}
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

createNativeStackNavigator

Фабрична функція, що повертає об'єкт з двома компонентами: `{ Navigator, Screen }`.

- використовує нативні API навігації платформ;
- забезпечує високу продуктивність і плавні анімації;
- підтримує системні жести та кнопку «Назад»;
- зберігає стан попередніх екранів у пам’яті.

```
import { createNativeStackNavigator } from '@react-navigation/native-stack';

const Stack = createNativeStackNavigator();
// Stack.Navigator - Контейнер
// Stack.Screen - Елемент
```

Stack.Navigator Props: initialRouteName

Визначає, який екран буде показано першим при завантаженні навігатора.

- Якщо проп не передано - береться **перший** `Stack.Screen` у списку.
- Якщо передано - ігнорує порядок і рендерить вказаний екран.

```
<NavigationContainer>
  {/* initialRouteName="Profile" змушує навігатор ігнорувати порядок
     і завантажити ProfileScreen першим.
  */}
  <Stack.Navigator initialRouteName="Profile" id={'main'}>

    {/* За замовчуванням завантажився б цей екран (бо він перший у списку) */}
    <Stack.Screen name="Login" component={LoginScreen} />

    <Stack.Screen name="Profile" component={ProfileScreen} />

  </Stack.Navigator>
</NavigationContainer>
```

Stack.Navigator Props: screenOptions

Реалізація каскадності стилів. Дозволяє задати налаштування для **всіх** екранів у цьому навігаторі.

```
<NavigationContainer>
  <Stack.Navigator
    // ГЛОБАЛЬНІ НАЛАШТУВАННЯ
    // Застосуються до Home ТА Profile автоматично
    screenOptions={{
      headerStyle: { backgroundColor: '#f4511e' }, // Помаранчевий фон
      headerTintColor: '#fff',                     // Білий колір тексту і стрілки
      headerTitleStyle: { fontWeight: 'bold' },    // Жирний шрифт
      headerTitleAlign: 'center',                  // Заголовок по центру (Android)
    }}
  >
    <Stack.Screen name="Home" component={HomeScreen} options={{ title: 'Головна сторінка' }} />
    <Stack.Screen name="Profile" component={ProfileScreen} options={{ title: 'Профіль' }} />
  </Stack.Navigator>
</NavigationContainer>
```

Stack.Navigator Props: id

Властивість `id` використовується для **ідентифікації навігаторів** у складних навігаційних схемах, де одночасно присутні кілька стеків.

```
<NavigationContainer>
  {/* Присвоюємо унікальний ID головному навігатору.*/}
  <Stack.Navigator id="GlobalStack" screenOptions={{headerShown: false}}>

    <Stack.Screen name="Login" component={LoginScreen} />

    {/* Тут ми вкладаємо один навігатор в інший */}
    <Stack.Screen name="AppFlow" component={NestedNavigator} />

  </Stack.Navigator>
</NavigationContainer>
```

Stack.Group (Logical Grouping)

Компонент-обгортка, який не впливає на UI-структуру (не додає View), але дозволяє групувати екрани для спільних налаштувань. **Use Case: Модальні вікна** Відокремлення звичайних екранів від модальних без створення вкладених навігаторів.

```
<NavigationContainer>
  <Stack.Navigator>

    {/* ГРУПА 1: Звичайні екрани */}
    <Stack.Group>
      <Stack.Screen name="Home" component={HomeScreen} options={{ title: 'Мій Додаток' }} />
      <Stack.Screen name="Details" component={DetailsScreen} />
    </Stack.Group>

    {/* ГРУПА 2: Модальні вікна*/}
    <Stack.Group screenOptions={{ presentation: 'modal' }}>
      <Stack.Screen name="Help" component={HelpModal} />
    </Stack.Group>

  </Stack.Navigator>
</NavigationContainer>
```

Stack.Screen: Name vs Component

Це два обов'язкові пропси.
1. `name` (String): Унікальний ключ маршруту. Використовується для навігації: `navigation.navigate('Profile')`.
2. `component` (Reference): Посилання на React-компонент.
Performance Warning: Ніколи не передавайте інлайн-функцію

```
// ❌ BAD: Створює новий компонент при кожному рендері. Втрата фокусу, лаги.
<Stack.Screen name="Home" component={() => <Home user={user} />} />

// ✅ GOOD: Передача посилання.
<Stack.Screen name="Home" component={Home} />
```

Stack.Screen Props: options

Налаштування для **конкретного** екрана. Має найвищий пріоритет (переписує `screenOptions`).
Може бути:

1. **Об'єктом:** Статичні налаштування.
2. **Функцією:** Доступ до `route` (params) та `navigation`.

```
<Stack.Screen
  name="Profile"
  component={ProfileScreen}
  options={({ route }) => ({
    title: `Профіль: ${route.params.name}`
  })}
/>
```

Stack.Screen Props: initialParams

Задає параметри за замовчуванням. Захищає від `undefined` у `route.params`. **Сценарій:** Ми відкриваємо екран "Профіль" без параметрів (з таббару).

```
<Stack.Screen
  name="Profile"
  component={ProfileScreen}
  initialParams={{ userId: 'me', mode: 'view' }}
/>

// У компоненті ProfileScreen:
// route.params.userId дорівнюватиме 'me', якщо ми не передали інше.
```

Stack.Screen Props: getId

`getId` — це функція для **унікалізації екземплярів одного й того самого екрана** в навігаційному стеку. Вона вирішує проблему навігації **на той самий екран із різними параметрами**, коли потрібно створити новий запис у стеку, а не просто оновити існуючий.
Навігація виду: `navigate('Product', { id: 1 })` → `navigate('Product', { id: 2 })`
За замовчуванням React Navigation може:

- оновити параметри поточного екрана `Product`;
- або проігнорувати навігацію (залежно від методу переходу).

У результаті **новий екран у стеку не створюється**.

```
<Stack.Screen
  name="Product"
  component={ProductScreen}
  getId={({ params }) => params.id} // Унікальний ID для кожного екрана
/>
```

Stack.Screen Props: navigationKey

Дозволяє вручну керувати ключем екрана в стейті навігації. Це проп для складних сценаріїв.
Якщо `navigationKey` зміниться, React Navigation вважатиме, що це зовсім інший екран, демонтує старий і змонтує новий (повний скидання стейту).
Використовується рідко, зазвичай коли потрібно примусово перезавантажити екран при зміні певних глобальних умов (наприклад, зміна мови або зміна користувача без виходу з екрана).

```
<NavigationContainer>
  <Stack.Navigator>
    <Stack.Screen
      name="Home"
      component={CounterScreen}
      navigationKey={keyId}
    />
  </Stack.Navigator>
</NavigationContainer>
```

Conditional Rendering (Auth Flow)

Неправильний підхід — виконувати перевірку в `useEffect` на Home Screen і програмно перенаправляти користувача на Login. У такому випадку користувач короткочасно бачить захищений екран, після чого відбувається редирект, а кнопка «Назад» дозволяє повернутися на Home, що є помилкою з точки зору безпеки та UX.

Правильний підхід — **conditional rendering навігаційних стеків**. Залежно від стану авторизації рендериться або стек авторизованого користувача, або стек входу. Захищені екрани для неавторизованого користувача взагалі не монтуються.

```
<Stack.Navigator>
  {isLoggedIn ? (
    // Екрани доступні тільки авторизованим
    <Stack.Group>
      <Stack.Screen name="Home" component={HomeScreen} />
      <Stack.Screen name="Profile" component={ProfileScreen} />
    </Stack.Group>
  ) : (
    // Екрани доступні тільки гостям
    <Stack.Group screenOptions={{ headerShown: false }}>
      <Stack.Screen name="SignIn" component={SignInScreen} />
      <Stack.Screen name="SignUp" component={SignUpScreen} />
    </Stack.Group>
  )}
</Stack.Navigator>
```

Методи навігації **(Imperative API)**

Для керування переходами **кожен екран отримує об’єкт `navigation`**.

Доступ до нього можливий двома способами. Перший — через `props.navigation`, який доступний лише компонентам, безпосередньо оголошеним у `Stack.Screen`. Другий — через хук `useNavigation()`, який можна використовувати з будь-якого вкладеного компонента, зокрема кнопок або допоміжних UI-елементів.

```
function MagicButton() {
  const navigation = useNavigation(); // <-- Хук

  return (
    <Button
      title="Кнопка з хуком (useNavigation)"
      color="green"
      onPress={() => navigation.navigate('Details')}
    />
  );
}
```

Базовий набір методів навігації:

`navigate()` — інтелектуальний перехід між екранами.
`push()` — примусове додавання нового екрана в стек.
`goBack()` — повернення на попередній екран.
`reset()` — повне переписування навігаційної історії.

Navigate vs Push (Архітектурна різниця)

1. `navigation.navigate('Profile')`

- **Логіка:** "Перейди на цей екран".
- **Поведінка:** Перевіряє історію. Якщо екран вже є в стеку — повертається до нього. Якщо немає — створює новий.
- **Кейс:** Таби, Налаштування, Меню.

2. `navigation.push('Profile')`

- **Логіка:** "Поклади нову картку зверху".
- **Поведінка:** Завжди створює нову інстанцію екрана.
- **Кейс:** Стрічка новин, Товари (Товар А -> Схожий Товар Б -> Знову Товар А).

Керування історією (Back Actions)

Навігація назад — це зняття верхнього елемента стеку (Pop).

Методи:

1. **`goBack()`**: Стандартна дія (аналог кнопки "Назад"). Безпечна (нічого не робить, якщо стек пустий).
2. **`pop(n)`**: Повернутися на n екранів назад.
3. **`popToTop()`**: Миттєве повернення до першого екрана в стеку.

Кейс `popToTop()`: Сценарій оформлення замовлення:

- Cart -> Checkout -> Payment -> Success Screen.
- На екрані "Success" кнопка "Done" має вести не на Payment, а на початок (Cart або Catalog).

Деструктивна навігація (Replace & Reset)

Іноді нам потрібно **заборонити** повернення назад.

1. **`replace('Home')`**

- Замінює поточний екран на новий.
- *Кейс:* Splash Screen -> Home. Користувач не повинен повернутися на заставку.

2. **`reset({ index: 0, routes: [...] })`**

- Повністю знищує поточний стан навігатора і створює новий.
- *Кейс:* **Logout**.

```
navigation.reset({
  index: 0,
  routes: [{ name: 'Login' }],
});
```

Передача даних (Passing Params)

Дані передаються другим аргументом у методах `navigate` або `push`.

```
<Button
  title="MacBook Pro"
  onPress={() => navigation.navigate( args: 'Details', {
    itemId: 101,
    name: 'MacBook Pro M3',
    price: '$2000'
  })}
/>
```

Динамічне оновлення (setParams)

Екран може змінювати свої параметри вже *після* монтування. Це критично для оновлення UI в хедері. **Кейс - Search Bar у заголовку:**

1. Хедер отримує текст із `route.params.query`.
2. Екрана містить Input.
3. При вводі ми оновлюємо params.

```
<TextInput
  style={{
    borderWidth: 1,
    borderColor: '#ccc',
    padding: 10,
    fontSize: 18,
    borderRadius: 8
  }}
  placeholder="Що шукаємо?"
  value={query}
  // 2. ОНОВЛЮЄМО параметри при вводі тексту
  // Це змусить навігатор перерендерити цей екран і оновити опції
  onChangeText={(text) => navigation.setParams( params: { query: text })}
/>
```

UI зсередини компонента **(setOptions)**

Кнопка в хедері (наприклад, "Save" або "Edit") повинна взаємодіяти зі стейтом компонента екрана.

Проблема: Хедер знаходиться "ззовні" компонента. **Рішення:** `navigation.setOptions`:

```
useLayoutEffect( effect: () => {

  navigation.setOptions({
    // Додаємо кнопку праворуч
    headerRight: () => (
      <Button
        title="Save"
        onPress={saveNote}
      />
    ),
  });

}, deps: [navigation, noteText]);
```

Перевірка фокусу (useIsFocused)

У React Navigation екрани **не розмонтовуються**, коли користувач переходить глибше в стек. Тому `useEffect` **не спрацьовує** при переходах.

Це означає, що код **не виконується автоматично при поверненні на екран**.

Рішення

Для визначення моменту, коли екран стає активним, використовується хук `useIsFocused`.

Він повертає `true`, коли екран у фокусі, і `false` — коли ні, зокрема при переході назад (`Back`).

```
import { useIsFocused } from '@react-navigation/native';

function Profile() {
  const isFocused = useIsFocused(); // true або false

  useEffect(() => {
    if (isFocused) {
      // Екран став активним (навіть при поверненні Back)
      analytics.logScreenView('Profile');
    }
  }, [isFocused]);
}
```

Події (Lifecycle Events)

Для більш кращого контролю можна підписуватися на події навігатора через `addListener`.

- **`focus`**: Екран з'явився (аналог `componentDidAppear`).
- **`blur`**: Екран зник (перейшли на інший таб або стек).
- **`state`**: Будь-яка зміна в навігаційному дереві.

```
useEffect(() => {
  const unsubscribe = navigation.addListener('focus', () => {
    // Оновити дані з сервера при поверненні
    refetchData();
  });

  return unsubscribe;
}, [navigation]);
```

UX Патерн: Prevent Going Back

Блокування кнопки "Назад", якщо у користувача є незбережені зміни. Використовується подія **`beforeRemove`**.

```
useEffect( effect: () => {
  // Підписуємося на подію спроби виходу з екрана
  const unsubscribe = navigation.addListener('beforeRemove', (e) => {

    if (!hasUnsavedChanges) {
      return;
    }

    e.preventDefault();

    Alert.alert(
      title: 'Відхилити зміни?',
      message: 'У вас є незбережений текст. Ви точно хочете вийти?',
      buttons: [
        { text: 'Залишитись', style: 'cancel', onPress: () => {} },
        {
          text: 'Вийти (Скинути)',
          style: 'destructive',
          onPress: () => navigation.dispatch(e.data.action),
        },
      ]
    );
  });

  return unsubscribe;
}, deps: [navigation, hasUnsavedChanges]);
```

Керування відображенням (headerShown)

Це **найпростіша, але одна з найчастіше використовуваних опцій навігації**.

Типовий сценарій - екрани авторизації, Splash Screen або повноекранні карти, де стандартний хедер не потрібен і заважає сприйняттю інтерфейсу.

Важливо: при приховуванні хедера **зникає системна кнопка «Назад»**. У такому випадку екран повинен мати **власний механізм виходу** — кнопку або жест, який явно керує навігацією.

```
{/* ПРИХОВУЄМО ХЕДЕР */}
<Stack.Screen
  name="Login"
  component={LoginScreen}
  options={{ headerShown: false }}
/>
```

Заголовок: **Text vs Component**

Текст у верхній частині екрана (header) можна змінити **двома способами**.

Перший - через `title`, який приймає звичайний рядок.
Це найпростіший варіант, що використовується:

- як заголовок у хедері;
- як fallback для табів;

```
options={{
  // Варіант А: Просто текст
  title: 'Мій Профіль',

  // Варіант Б: Логотип (переписує title)
  headerTitle: () => <LogoImage width={100} height={30} />
}}
```

Другий - через `headerTitle`, який дозволяє передати **власний React-компонент**. Цей підхід дає повну кастомізацію: логотип, поле пошуку, селектор або будь-який складний UI.

Стилізація Контейнера (headerStyle)

- **`headerStyle`**: Об'єкт стилів для контейнера (Background Color, Height).
- **`headerTintColor`**: Колір **усього** контенту всередині (текст заголовка, стрілка назад, кнопки).

```
<Stack.Navigator
  screenOptions={{
    headerStyle: { backgroundColor: '#f4511e' },
    headerTintColor: '#fff', // Білий текст і стрілки
    headerTitleStyle: { fontWeight: 'bold' },
  }}
>
```

Типографіка та Вирівнювання

Android та iOS мають різну типографіку та вирівнювання.

- **iOS:** Заголовок по центру, жирний шрифт.
- **Android:** Заголовок зліва (Material Design).

Уніфікація (`headerTitleAlign`): Якщо дизайн вимагає однакового вигляду:

```
options={{
  headerTitleAlign: 'center', // Примусово центрувати на Android
  headerTitleStyle: {
    fontFamily: 'CustomFont-Bold',
    fontSize: 20
  }
}}
```

Кнопки (headerRight / headerLeft)

Навігаційна панель може містити **інтерактивні елементи**, такі як кнопки *Save*, *Menu* або *Share*.

Для цього використовуються властивості `headerRight` та `headerLeft`. Вони приймають **функцію, яка повертає React-компонент**, що рендериться відповідно з правого або лівого боку хедера.

```
<Stack.Navigator>
  <Stack.Screen
    name="Home"
    component={HomeScreen}
    options={{
      title: 'Головна',
      // 1. Кнопка ЗЛІВА (Замінює місце стрілки "Назад")
      headerLeft: () => (
        <Button
          title="≡ Menu"
          color="black"
          onPress={() => Alert.alert( title: 'Меню',  message: 'Відкриваємо бокову панель...')}
        />
      ),
      // 2. Кнопка СПРАВА (Дії)
      headerRight: () => (
        <Button
          title="Info ℹ"
          onPress={() => Alert.alert( title: 'Інформація',  message: 'Це версія додатку 1.0')}
        />
      ),
    }}
  />
```

Примусове приховування **Back (headerBackVisible)**

Іноді стек не пустий, але ми хочемо заборонити користувачу повертатися назад через UI (наприклад, після успішної оплати, поки йде анімація).

```
<Stack.Screen
  name="Success"
  component={SuccessScreen}
  options={{ headerBackVisible: false }}
/>
```

Користувач не побачить стрілку, але фізична кнопка "Назад" на Android все ще працюватиме (її треба блокувати окремо через `BackHandler`).

Прозорість та **Blur (headerTransparent)**

Ефект, коли контент (наприклад, картинка) заходить *під* хедер.

1. `headerTransparent: true`

- Хедер стає абсолютно позиціонованим (`position: absolute`).
- **Важливо:** Контент екрана зсувається вгору на висоту хедера. Потрібно додати `paddingTop`, якщо це не картинка.

2. `headerBlurEffect` (iOS only):

- Додає нативний `UIBlurEffect` (матове скло) на фон хедера.

Фон Екрана (contentStyle)

Поширена помилка - обгортати кожен екран у контейнер з `flex: 1` і фоновим кольором.

Такий підхід збільшує вкладеність компонентів і ускладнює структуру інтерфейсу без реальної потреби.

Правильне рішення - задавати фоновий стиль **на рівні навігатора** через `contentStyle`.
Це дозволяє централізовано керувати фоном екранів і зменшити глибину DOM-дерева. **Стилі макета мають належати навігації, а не кожному окремому екрану.**

```
<Stack.Navigator
  screenOptions={{
    contentStyle: { backgroundColor: '#FFFFFF' }
  }}
>
```

Анімації Переходів

Типи:

- **`default`**: Платформна (Slide on iOS, Fade/Reveal on Android).
- **`slide_from_bottom`**: Екран виїжджає знизу (стандарт Android 9+).
- **`fade`**: Плавне розчинення (добре для галерей).
- **`none`**: Миттєве перемикання (без анімації).

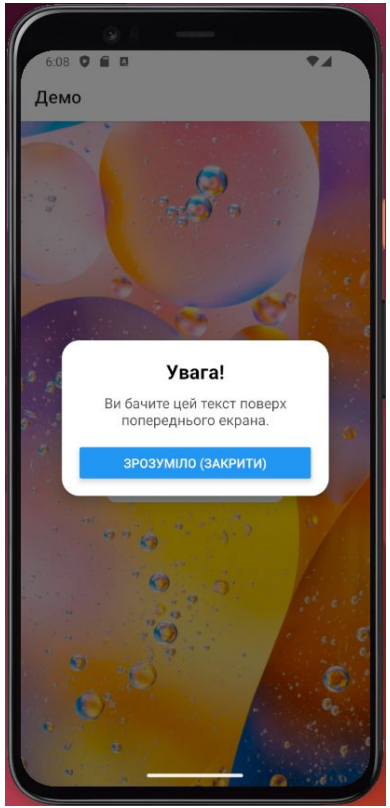
```
<Stack.Screen
  name="Fade"
  component={DemoScreen}
  options={{ animation: 'fade' }}
/>
```

Прозорі екрани

`transparentModal` дозволяє бачити попередній екран крізь новий.

Вимоги:

1. `presentation: 'transparentModal'`.
2. `animation: 'fade'`.
3. Фон самого компонента екрана має бути напівпрозорим (`rgba(0,0,0,0.5)`), інакше він буде білим.



Tab Navigator: Концепція

Tab Navigator - це навігація у вигляді **вкладок, розташованих у нижній частині екрана**.
Вона використовується для швидкого перемикання між **основними, рівноправними розділами застосунку**.

Кожна вкладка має власний навігаційний контекст і зазвичай містить **власний Stack Navigator**.

```
npx expo install @react-navigation/bottom-tabs
```

```
const Tab = createBottomTabNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Tab.Navigator id={'main-tabs'}>
        <Tab.Screen name="Home" component={HomeScreen} options={{ title: 'Головна' }}/>
        <Tab.Screen name="Settings" component={SettingsScreen} options={{ title: 'Опції' }}/>
      </Tab.Navigator>
    </NavigationContainer>
  );
}
```

tabBarIcon

`tabBarIcon` — це **ключовий елемент UI Tab Navigator**. Він приймає функцію рендеру, яка викликається автоматично залежно від стану вкладки.

Функція отримує параметри:

- `focused` - `true`, якщо вкладка активна;
- `color` - колір з теми навігації (active / inactive);
- `size` - рекомендований розмір іконки.

Це дозволяє:

- змінювати вигляд іконки залежно від активного стану;
- забезпечити узгодженість з темою застосунку;
- реалізувати зрозумілий та інформативний UI вкладок.

```
<Tab.Navigator
  screenOptions={({ route }) => ({
    tabBarIcon: ({ focused, color, size }) => {
      let iconName;

      if (route.name === 'Home') {
        iconName = focused ? 'home' : 'home-outline';
      } else if (route.name === 'Settings') {
        iconName = focused ? 'settings' : 'settings-outline';
      }

      return <Ionicons name={iconName} size={size} color={color} />;
    },
    headerShown: false,
  })}
>
```

Кольорова палітра

Керування кольорами активного та неактивного станів глобально для всього таббару.

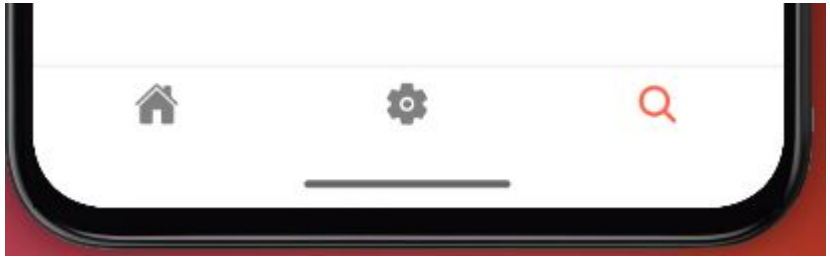
```
<Tab.Navigator
  screenOptions={({ route }) => ({
    tabBarActiveTintColor: 'blue',
    tabBarInactiveTintColor: 'gray',
  })}
>
```

tabBarLabel

Іноді заголовок у хедері та підпис під іконкою мають бути різними.

- **`title`**: Встановлює текст і для хедера, і для таба.
- **`tabBarLabel`**: Тільки під іконкою.
- **`tabBarShowLabel`**: Приховати текст повністю (дизайн "тільки іконки").

```
<Tab.Screen
  name="Search"
  component={Screen}
  options={{
    title: 'Пошук',
    tabBarShowLabel: false,
    tabBarIcon: ({ color }) => <Ionicons name="search" size={24} color={color} />
  }}
/>
```

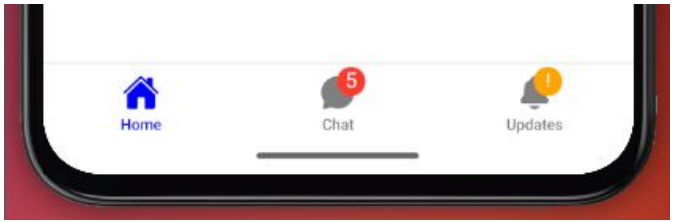


Бейджі та Сповіщення (tabBarBadge)

Червоні індикатори з лічильниками (*badges*), як у месенджерах, використовуються для привернення уваги до вкладки.
`tabBarBadge` задає значення бейджа — число або рядок. Якщо значення `null`, бейдж не відображається.
`tabBarBadgeStyle` відповідає за зовнішній вигляд бейджа: колір фону, розмір і стиль шрифту.

```
{/* 1. Звичайна вкладка без бейджа */}
<Tab.Screen name="Home" component={Screen} />

{/* 2. tabBarBadge: Число (int)
    Стандартний червоний бейдж з цифрою 5.
*/}
<Tab.Screen
  name="Chat"
  component={Screen}
  options={{
    tabBarBadge: 5
  }}
/>
```



Стилізація Бару (tabBarStyle)

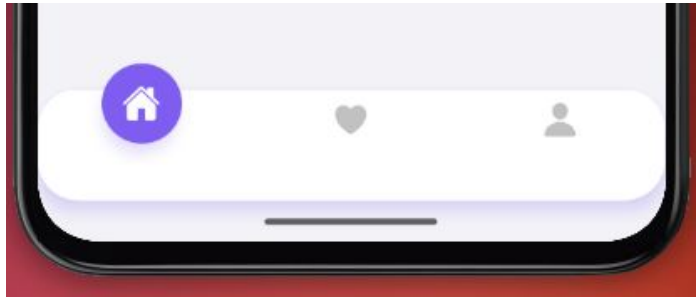
```
<NavigationContainer>
  <Tab.Navigator
    screenOptions={({ route }) => ({
      headerShown: false,
      tabBarShowLabel: false, // Прибираємо текст

      tabBarStyle: {
        position: 'absolute',
        bottom: 25,
        left: 20,
        right: 20,
        elevation: 0, // Вимикаємо стандартну тінь Android
        backgroundColor: '#ffffff',
        borderRadius: 25, // Сильніше заокруглення
        height: 70,
        ...styles.shadow,
      },
```

`tabBarStyle` — це властивість **Tab Navigator**, яка визначає зовнішній вигляд нижньої навігаційної панелі.
Вона дозволяє:

- задавати фон і прозорість панелі;
- керувати висотою та внутрішніми відступами;
- вмикати або прибирати тінь / бордер;
- адаптувати вигляд табів під дизайн застосунку.

Стилі застосовуються **глобально до всіх вкладок**, що забезпечує єдиний UI та спрощує підтримку.



Кастомні Кнопки (tabBarButton)

`tabBarButton` дозволяє **повністю замінити стандартну кнопку вкладки** власним React-компонентом.

Цей підхід використовується, коли стандартний вигляд або поведінка таба не підходять під дизайн чи UX-вимоги.

Типові сценарії:

- центральна кнопка дії (FAB);
- нестандартна форма або анімація;
- власна логіка обробки натискань.

Важливий момент: кастомна кнопка повинна **самостійно викликати навігацію**, оскільки стандартна поведінка замінюється.

```
<Tab.Screen
  name="Home"
  component={Screen}
  options={{
    tabBarButton: (props) => (
      <TouchableOpacity
        {...props}
        style={{
          flex: 1,
          backgroundColor: 'red',
          justifyContent: 'center',
          alignItems: 'center'
        }}
      >
        <Text style={{
          color: 'white',
          fontWeight: 'bold'
        }}>BUTTON</Text>
      </TouchableOpacity>
    )
  }}
/>
```

Blur-ефект (tabBarBackground)

Для створення ефекту **напівпрозорого, розмитого фону** таб-бару в стилі iOS використовується blur-ефект.

Такий підхід:

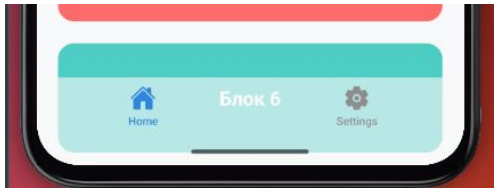
- зберігає видимість контенту під навігацією;
- додає глибину інтерфейсу;
- відповідає нативним iOS UI-патернам.

```
tabBarBackground: () => (
  <BlurView
    tint="light"      // Варіанти: 'light', 'dark', 'default'
    intensity={80}    // Сила розмиття (0-100)
    style={StyleSheet.absoluteFill} // Розтягуємо на весь бар
  />
),
```

Для реалізації blur-ефекту необхідно встановити бібліотеку:

```
npx expo install expo-blur
```

Вона надає компонент `BlurView`, який використовується як фон для таб-бару.



Менеджмент Пам'яті

За замовчуванням Таби тримають усі відкриті екрани в пам'яті. Це добре для швидкості, але погано для RAM.

Опція `unmountOnBlur: true`:

- **Поведінка:** Коли ви йдете з вкладки, її компонент знищується. Коли повертаєтесь - монтується заново.
- **Кейс:** Важкі екрани (Карти, Камера), які не потрібні у фоні.

Опція `freezeOnBlur: true`:

- **Поведінка:** Екран залишається в пам'яті, але *перестає рендеритися* (економить CPU).
- **Кейс:** Стандартна поведінка для більшості додатків (увімкнено за замовчуванням у нових версіях `react-native-screens`).

Ліниве завантаження **(lazy)**

- **`lazy: true` :** Екрани табів (наприклад, Профіль, Налаштування) не рендеряться, доки користувач на них не натисне.
- **`lazy: false`:** Усі екрани рендеряться одночасно при запуску додатка.

Клавіатура (tabBarHideOnKeyboard)

`tabBarHideOnKeyboard` — це опція Tab Navigator, яка **автоматично приховує нижню навігаційну панель**, коли на екрані з'являється клавіатура.

Це особливо важливо для:

- форм введення;
- чатів;
- екранів з активним текстовим вводом.

```
<Tab.Navigator
  screenOptions={{
    tabBarHideOnKeyboard: true
  }}
>
```

Увімкнення цієї опції:

- запобігає перекриттю полів вводу таб-баром;
  зменшує візуальний шум;
- покращує UX на невеликих екранах.

Перехоплення натискань (tabPress)

`tabPress` — це подія Tab Navigator, яка дозволяє **перехоплювати натискання на вкладку** до виконання стандартної навігації.

Використовується, коли потрібно:

- виконати додаткову логіку перед переходом;
- заблокувати навігацію за умовою;
- замінити стандартну поведінку власною дією.

За замовчуванням натискання на вкладку **перемикає активний екран**.
Обробка `tabPress` дозволяє скасувати цю дію або доповнити її.

```
<Tab.Screen
  name="Profile"
  component={Screen}
  initialParams={{ name: 'Особистий кабінет' }}

  listeners={{
    tabPress: (e) => {
      if (!isLoggedIn) {
        // 1. Зупиняємо стандартний перехід на вкладку
        e.preventDefault();

        // 2. Виконуємо свою логіку (Alert)
        Alert.alert(
          title: "Доступ заборонено",
          message: "Будь ласка, увійдіть у систему, щоб побачити профіль.",
          buttons: [{ text: "ОК" }]
        );
      }
    },
  }}
/>
```

Scroll to Top (Native Behavior)

Стандартний патерн iOS/Android: повторне натискання на активний таб має скролити список вгору. Це **не працює** автоматично . Потрібно реалізовувати вручну через `useScrollToTop` hook.

```
import { useScrollToTop } from '@react-navigation/native';

function FeedScreen() {
  const ref = React.useRef(null);
  useScrollToTop(ref); // Слухає клік по табу

  return <FlatList ref={ref} ... />;
}
```

Довге натискання (tabLongPress)

```
listeners={{
  tabLongPress: (e) => {
    showDebugMenu();
  },
}}
```

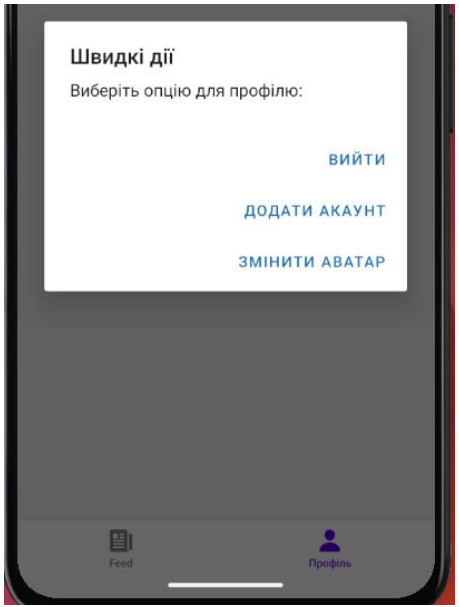
`tabLongPress` — це подія Tab Navigator, яка спрацьовує **при довгому натисканні на вкладку**.

Використовується для реалізації **додаткових або контекстних дій**, не перевантажуючи основний інтерфейс.

Типові сценарії:

- відкриття контекстного меню;
  швидкі дії (shortcut-функції);
- сервісні або приховані можливості.

Подія не замінює стандартну навігацію, а **доповнює її альтернативною взаємодією**.

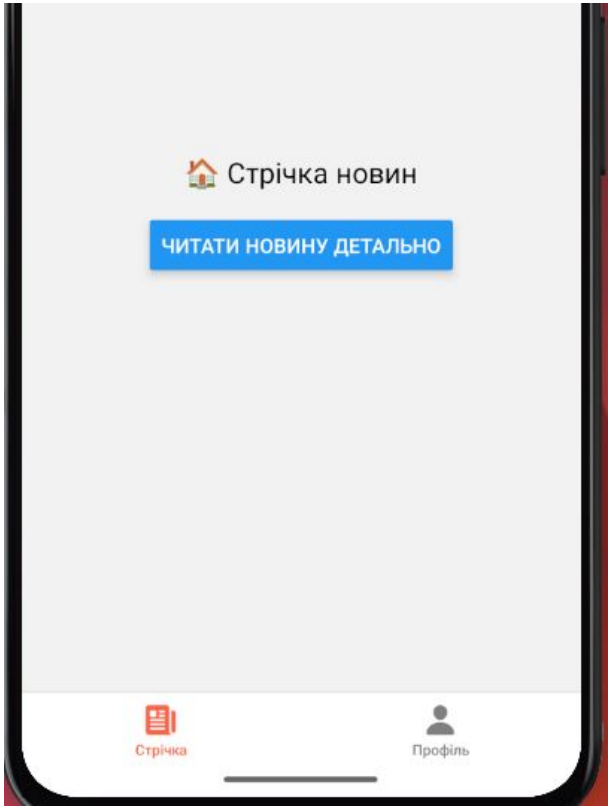


Вкладеність: **Stack in Tab**

Це **найпоширеніша структура мобільного застосунку**.

Базове правило:
Tab Navigator є батьківським навігатором, а Stack Navigator - вкладеним усередині кожної вкладки.

Кожна вкладка представляє окремий розділ застосунку та має власний навігаційний стек.



Drawer Navigator

Drawer Navigator — це тип навігації з **бічним меню**, яке відкривається жестом або кнопкою (зазвичай зліва).

Він використовується для доступу до **другорядних або сервісних розділів**, які не повинні постійно займати місце в основному UI.

Типові сценарії:

- Налаштування
- Профіль користувача

Drawer Navigator зазвичай є **обгорткою для Tab або Stack Navigator**, або використовується як **глобальне меню застосунку**.

Він не призначений для частих переходів між основними екранами.

Drawer Navigator

Для використання Drawer Navigator необхідно встановити сам навігатор:

```
npx expo install @react-navigation/drawer
```

Також потрібні додаткові залежності (обов’язково):

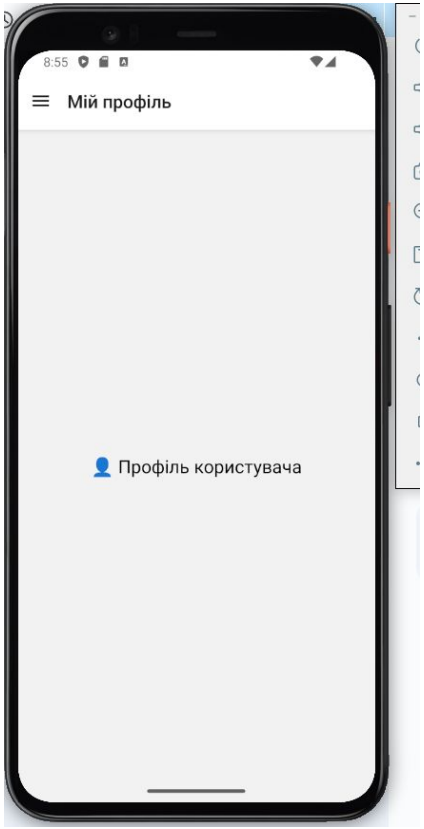
```
npx expo install react-native-gesture-handler react-native-reanimated
react-native-worklets react-native-screens
react-native-safe-area-context
```

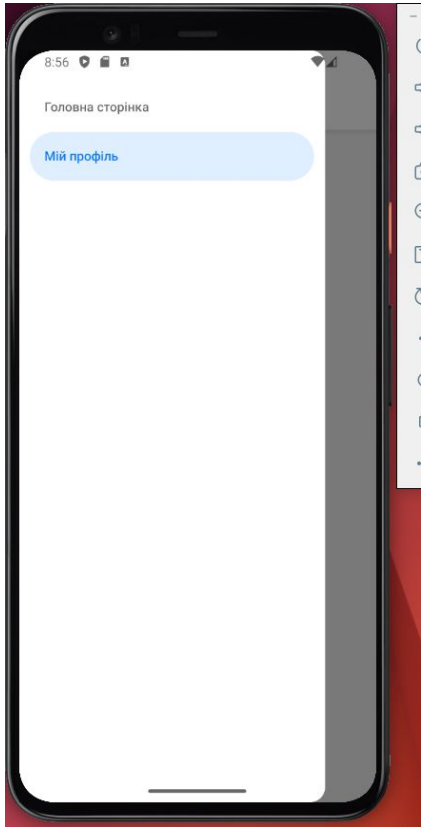
Після встановлення застосунок **потрібно перезапустити**.

Drawer Navigator

```
const Drawer = createDrawerNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Drawer.Navigator
        initialRouteName="Home"
      >
        <Drawer.Screen
          name="Home"
          component={HomeScreen}
          options={{ title: 'Головна сторінка' }}
        />
        <Drawer.Screen
          name="Profile"
          component={ProfileScreen}
          options={{ title: 'Мій профіль' }}
        />
      </Drawer.Navigator>
    </NavigationContainer>
  );
}
```





Типи анімацій (drawerType)

Drawer Navigator підтримує кілька режимів відкриття бічного меню, які впливають на поведінку основного екрана та сприйняття інтерфейсу.

`front` (Standard)
Меню відкривається поверх контенту як overlay. Основний екран залишається нерухомим.
Це стандартний і найпоширеніший режим.

`slide` (iOS-style)
Меню відкривається, **зсуваючи основний екран убік**.
Створює відчуття фізичної взаємодії з інтерфейсом.

`back`
Меню розташоване на задньому плані, а екран від'їжджає вбік, поступово відкриваючи його.
Візуально підкреслює ієрархію: контент «над» меню.

`permanent`
Меню завжди відображається на екрані.
Типовий сценарій — **планшетна версія (iPad) у ландшафтній орієнтації** або desktop-like layout.

```
<Drawer.Navigator
  screenOptions={{
    drawerType: isTablet ? 'permanent' : 'slide',
  }}
>
```

Позиція та Стиль **(drawerPosition, drawerStyle)**

Позиціонування

Положення бічного меню визначається властивістю `drawerPosition`.

- `left` — стандартне розташування
- `right` — альтернативне розташування

Важливо (Android):

Системний жест «Назад» (свайп від лівого краю) може конфліктувати з лівим Drawer.

У таких випадках доцільно:

- змінити позицію меню на `right`, або
- обмежити зону жесту відкриття.

Розміри та стиль

За замовчуванням Drawer займає близько **75% ширини екрана**.

Через `drawerStyle` можна централізовано керувати його зовнішнім виглядом.

Типові налаштування:

- фон меню;
- фіксована або адаптивна ширина;

```
<NavigationContainer>
  <Drawer.Navigator
    screenOptions={{
      drawerPosition: 'right',
      drawerStyle: {
        backgroundColor: '#c6cbef',
        width: 240,  // Фіксована ширина
      },
    }}
  >
    <Drawer.Screen name="Home" component={Screen} />
    <Drawer.Screen name="Profile" component={Screen} />
    <Drawer.Screen name="Settings" component={Screen} />
  </Drawer.Navigator>
</NavigationContainer>
```

Конфлікт Жестів (swipeEnabled)

Критичний момент для додатків з картами (Google Maps) або горизонтальними скролами.

Проблема: Користувач хоче посунути карту вправо, а замість цього відкриває меню.

Рішення: Вимикати жест відкриття на конкретних екранах. Відкривати меню тільки кнопкою-"бургером".

```
<Drawer.Screen
  name="Map"
  component={MapScreen}
  options={{ swipeEnabled: false }} // Тільки кнопка
/>
```

Кастомний Контент (drawerContent)

`drawerContent` дозволяє **повністю замінити стандартний вміст бічного меню** власним React-компонентом.

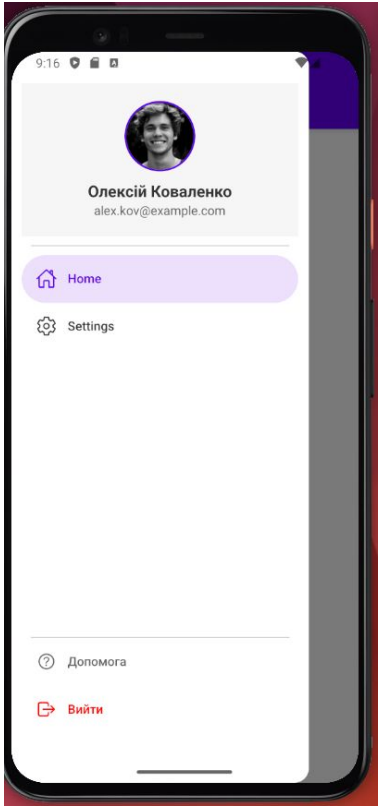
Це використовується, коли стандартний список екранів не відповідає вимогам дизайну або UX.

Типові сценарії:

- профіль користувача з аватаром;
- кастомні пункти меню та секції;

Основні пропси, які доступні в `drawerContent`

- `navigation` — об’єкт навігації (navigate, closeDrawer, toggleDrawer)
  `state` — поточний стан Drawer (активний маршрут, список екранів)
- `descriptors` — конфігурація екранів (options, labels, icons)
- `progress` — анімоване значення відкриття Drawer (для кастомних анімацій)



Візуальні ефекти **(overlayColor)**

`overlayColor` визначає **колір та прозорість затемнення основного контенту**, коли Drawer відкритий.

За замовчуванням використовується напівпрозорий чорний фон: `rgba(0, 0, 0, 0.5)`, який візуально відокремлює меню від контенту.

```
screenOptions={{
  overlayColor: 'transparent',
  sceneContainerStyle: { backgroundColor: 'white' }
}}
```

Програмне керування

Бічне меню можна відкривати та закривати **програмно**, наприклад, кнопкою в хедері.

Для цього використовуються методи об’єкта `navigation`:

- `navigation.openDrawer()` — відкрити меню
- `navigation.closeDrawer()` — закрити меню
- `navigation.toggleDrawer()` — перемкнути стан меню

```
<View style={styles.btnContainer}>
  <Button
    title="Відкрити меню (Open)"
    onPress={() => navigation.openDrawer()}
  />
</View>

<View style={styles.btnContainer}>
  <Button
    title="Перемкнути (Toggle)"
    color="orange"
    onPress={() => navigation.toggleDrawer()}
  />
</View>
```

Статус

Хук `useDrawerStatus` дозволяє визначити, **чи відкритий Drawer у поточний момент**. Він повертає одне з двох значень:

- `'open'`
- `'closed'`

Це корисно для:

- умовного рендерингу UI;
- синхронізації кнопок і анімацій;
- кастомної поведінки хедера.

Підсумок: Навігація в **React Native**

Навігація в React Native — це **частина архітектури застосунку**, а не просто перемикання між екранами.
Вона керує станом, життєвим циклом екранів і користувацькими потоками.

Ключові принципи

- Навігація ≠ routing → **навігація = state management**
- Архітектура навігації визначає UX і масштабованість
- Екрани не знищуються автоматично — вони керуються стеком
- Авторизація повинна **визначати навігацію**, а не виконуватись після рендеру

Основні типи навігаторів

- **Stack Navigator** - послідовні сценарії (Details, Forms, Checkout)
- **Tab Navigator** - паралельні розділи (Feed, Search, Profile)
- **Drawer Navigator** - сервісні та другорядні екрани

У реальних застосунках навігатори **комбінуються**.

Хук **useRoute**

`useRoute` — це хук React Navigation, який надає **доступ до поточного маршруту екрана**.
Він використовується для отримання:

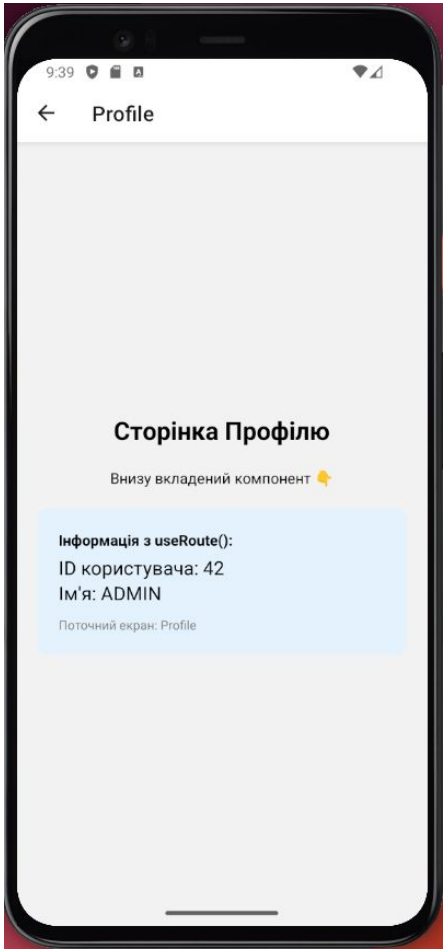
- параметрів навігації (`route.params`);
- імені поточного екрана;
- інформації про маршрут **без прокидування пропсів**.

`useRoute` доцільний, коли:

- параметри потрібні у вкладеному компоненті;
- екран має складну внутрішню структуру;
- ви хочете уникнути `props drilling`.

Що повертає

- `route.name` — ім’я поточного екрана
- `route.params` — передані параметри навігації



Хук **useNavigationState**

`useNavigationState` — це хук React Navigation, який надає **доступ до поточного навігаційного стану**.
Він дозволяє читати:

- активний маршрут;
- індекс поточного екрана;
- повну структуру навігаційного стеку або табів.

`useNavigationState` застосовується, коли потрібно:

- визначити, **на якому екрані зараз користувач**;
- змінювати UI залежно від активного маршруту;
- реалізувати умовну логіку на основі навігаційної структури;
- синхронізувати навігацію з аналітикою або глобальним станом.

Що повертає

Хук приймає selector-функцію і повертає вибрану частину навігаційного стану, наприклад:

- поточний `index`;
  масив `routes`;
- активний `route.name`.

Хук **useIsFocused (UI Trigger)**

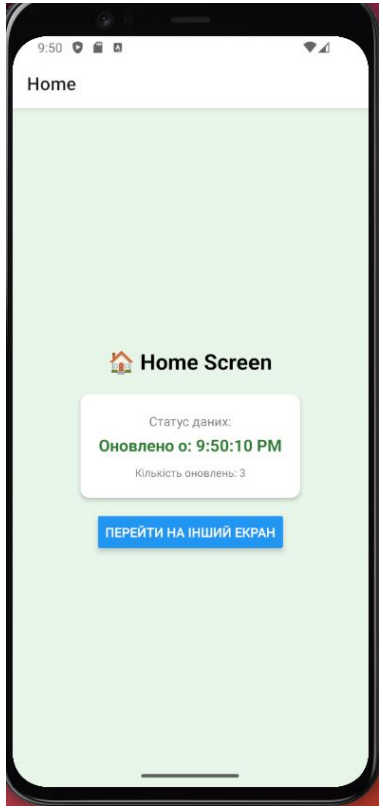
`useIsFocused` — це хук React Navigation, який дозволяє визначити, **чи знаходиться екран у фокусі в поточний момент**.

На відміну від `useEffect`, екран у навігації **не розмонтовується** при переходах, тому фокус ≠ монтування.

Коли використовувати

`useIsFocused` застосовується, коли потрібно:

- виконати дію при поверненні на екран;
- оновити дані після переходу `Back`;



Хук **useFocusEffect (Side Effects)**

`useFocusEffect` — це хук React Navigation, який дозволяє **виконувати побічні ефекти тільки тоді, коли екран перебуває у фокусі**. Він поєднує ідею `useEffect` з навігаційним фокусом екрана.

Коли використовувати

`useFocusEffect` застосовується, коли потрібно:

- запускати логіку **при кожному вході на екран**;
- очищати ресурси при втраті фокусу;
- обробляти події, прив’язані до видимості екрана;
- уникнути ручної перевірки `useIsFocused`.

Як працює

- ефект виконується, коли екран отримує фокус;
- cleanup-функція викликається, коли екран втрачає фокус;
- екран не розмонтовується між переходами.

Callback у `useFocusEffect` **має бути мемоізований** (`useCallback`), інакше ефект виконуватиметься некоректно.

Подія **state**

Спрацьовує при **будь-якій** зміні навігаційного стану (навіть якщо це сталося в іншому навігаторі).

Кейс: Глобальна аналітика Відстеження шляху користувача по всьому додатку.

```
onStateChange={async () => {
  const previousRouteName = routeNameRef.current;

  const currentRoute = navigationRef.getCurrentRoute();
  const currentRouteName = currentRoute.name;

  if (previousRouteName !== currentRouteName) {
    AnalyticsService.logScreenView(currentRouteName);
  }

  routeNameRef.current = currentRouteName;
}}
```