Розробка мобільних додатків

Лекція 8 - Expo Router

Обмеження класичних підходів до навігації

1. **Надмірний Boilerplate:** Кожен маршрут потребує ручного імпорту та реєстрації.
    - *Наслідок:* Гігантські файли конфігурації, складні для підтримки та рев'ю.
2. **Структурний розрив:** Відсутність зв’язку між розташуванням файлу та URL-адресою.
    - *Наслідок:* Ускладнена навігація по коду в масштабних проєктах.
3. **Складність Deep Linking:** Необхідність вручну синхронізувати об'єкти конфігурації із зовнішніми URL.
    - *Наслідок:* Висока ймовірність помилок («биті посилання») при кожному рефакторингу.
4. **Слабкий Web-Native Sync:** Неприродна робота з адресою рядка браузера та нульова SEO-орієнтованість.
    - *Наслідок:* Додаток у вебі не відчувається як повноцінний сайт.
5. **Розрив логіки та стану:** Складне керування параметрами та станом при переходах між незалежними навігаторами.

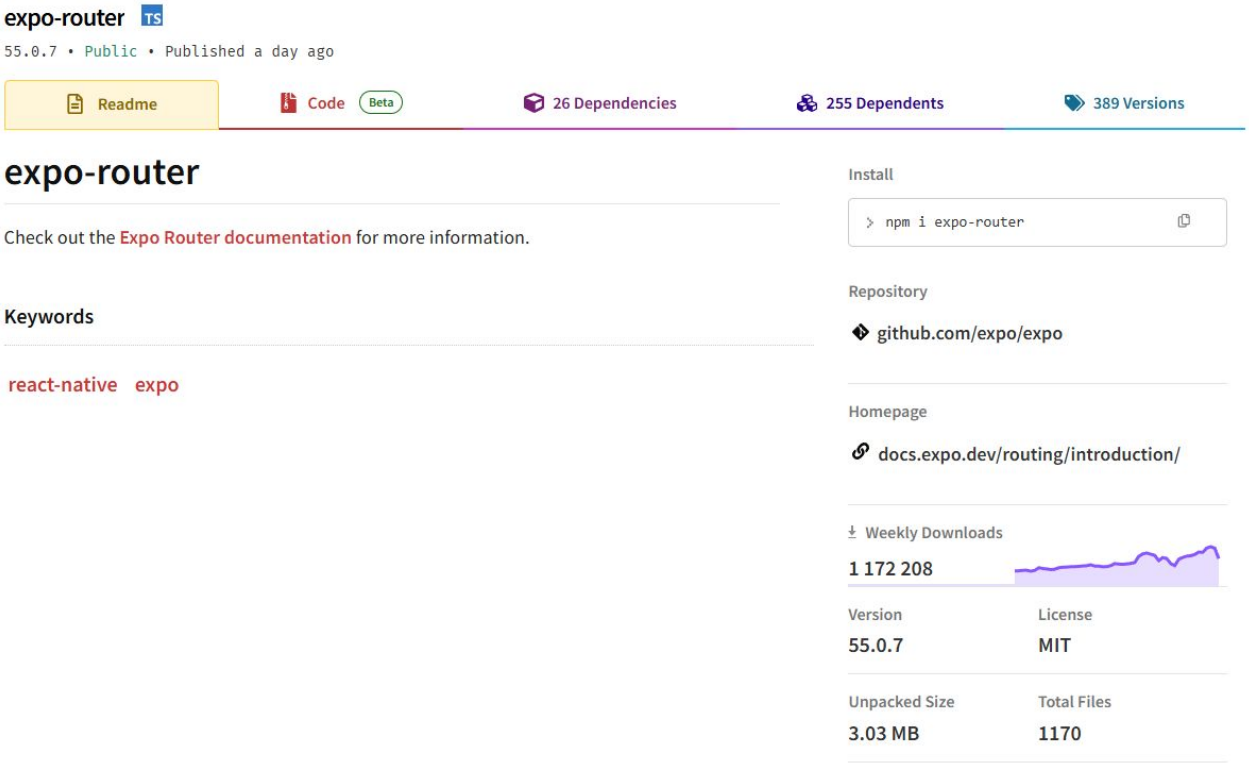
Expo Router

Expo Router — це сучасна бібліотека маршрутизації, яка переносить філософію Next.js у мобільну розробку. Вона дозволяє створювати додатки, що працюють однаково на iOS, Android та Web, використовуючи єдину систему навігації.

Переваги:

1. **Автоматизація Deep Linking:** Кожен екран автоматично отримує свою URL-адресу.
2. **Native Runtime:** Попри «веб-подібний» синтаксис, використовуються нативні компоненти.
3. **Refactoring Safety:** Перейменування файлу автоматично змінює маршрут у всьому додатку.
4. **Bundle Splitting:** Автоматична оптимізація розміру додатка на основі використовуваних роутів.

Expo Router


expo-router
55.0.7 • Public • Published a day ago
Readme
Code
Beta
26 Dependencies
255 Dependents
389 Versions
expo-router
Check out the Expo Router documentation for more information.
Keywords
react-native
expo
Install
npm i expo-router
Repository
github.com/expo/expo
Homepage
docs.expo.dev/routing/introduction/
Weekly Downloads
1 172 208
Version
55.0.7
License
MIT
Unpacked Size
3.03 MB
Total Files
1170

Expo Router

Версія 1 (2022)

- перша реалізація file-based маршрутизації
- експериментальний статус
- обмежена стабільність та функціональність

Версія 2 (2023)

- офіційно стабільний реліз
- інтеграція з Expo SDK
- підтримка:
  - layouts (`_layout`)
  - вкладених маршрутів
  - dynamic routes (`[id]`)
- покращена підтримка deep linking

Версії 3-6 (2024–2025)

- уніфікація підходу для **mobile + web**
- покращення продуктивності
- розвиток SSR / web-напрямку
- спрощення API та покращення DX

Концепція директорії **/app: File System** як **Source of Truth**

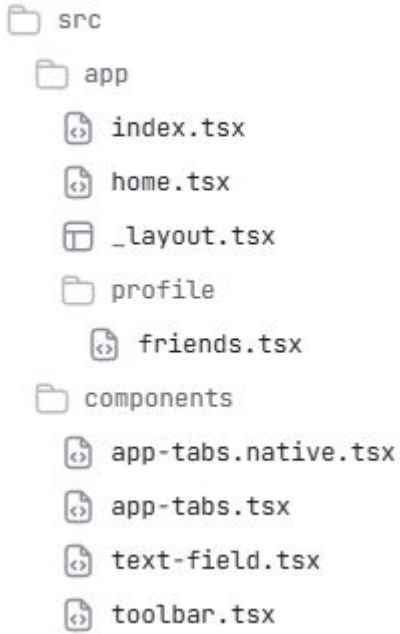
В Expo Router файлова структура проекту є **маніфестом** навігації.
Це реалізує принцип Єдине джерело істини.

Механізм мапінгу:

- Директорія `app` є коренем навігаційного графа.
- Кожен файл `.tsx`, `.ts`, `.js` у цій директорії стає окремим маршрутом.
- Вкладені папки створюють вкладені шлях (сегменти URL).

Переваги підходу:

- Нові розробники розуміють навігацію, просто глянувши на дерево файлів.
- Виключена можливість ситуації, коли файл існує, але не зареєстрований у навігаторі.



Native + Web: Концепція **Universal Routing**

Як це працює:

1. **На мобільних пристроях:** Маршрут `/user/settings` ініціює нативний перехід у стеку з підтримкою нативних жестів «back».
2. **У браузері (Web):** Той самий маршрут змінює URL у адресному рядку браузера та підтримує стандартне керування історією.

Технічні наслідки:

- **SEO:** Веб-версія додатка стає повністю індексованою пошуковими системами.
- **Shareability:** Користувач може скопіювати посилання на будь-який екран додатка і надіслати іншому користувачу; при відкритті додаток автоматично перейде до потрібної глибини стека.

Встановлення

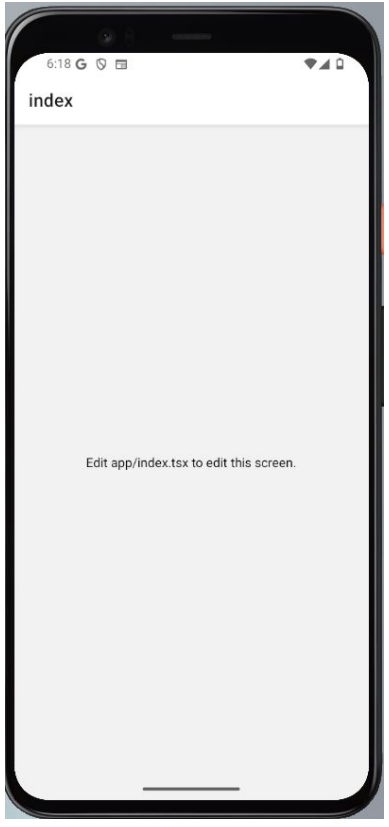
Новий проект: Використання офіційного шаблону, що містить попередньо налаштовану структуру:
npx create-expo-app@latest

Інтеграція в існуючий проект: Необхідна ручна інсталяція: `expo-router`, `react-native-safe-area-context`, `react-native-screens`, `expo-linking`.

Manual installation - Expo Documentation

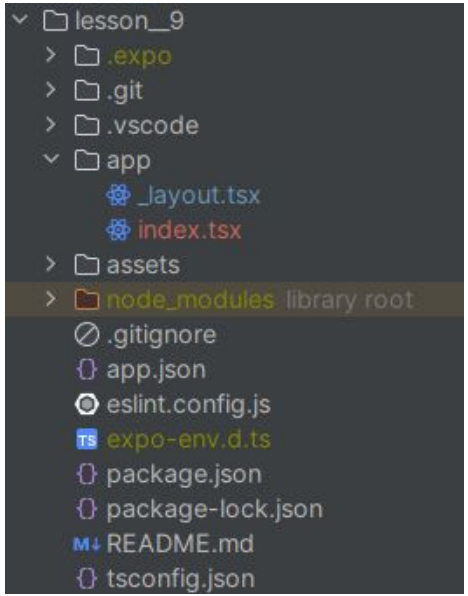
npm run reset-project

```
$ npm run reset-project

> lesson__9@1.0.0 reset-project
> node ./scripts/reset-project.js

Do you want to move existing files to /app-example instead of deleting them? (Y/n): n
❌/app deleted.
❌/components deleted.
❌/hooks deleted.
❌/constants deleted.
❌/scripts deleted.

📁 New /app directory created.
📄 app/index.tsx created.
📄 app/_layout.tsx created.

✅Project reset complete. Next steps:
1. Run `npx expo start` to start a development server.
2. Edit app/index.tsx to edit the main screen.
```



Анатомія директорії /app

Директорія `/app` — це не просто папка з компонентами, а **декларативна карта маршрутів**.



_layout.tsx як архітектурний патерн

Файл `_layout.tsx` виконує роль **Middleware** або **Wrapper** для всіх маршрутів у поточній та вкладених директоріях.

Функціональні обов'язки Layout:

1. Обгортка екранів у React Context.
2. **Shared UI:** Визначення постійних елементів інтерфейсу.
3. **Navigation Logic:** Вибір типу навігації .

```tsx
import { Stack } from "expo-router";

no usages  Sergey *
export default function RootLayout() {
  return <Stack />;
}
```

Навіщо потрібні **Layouts?**

У звичайній розробці кожен екран часто змушений самостійно рендерити хедер або таб-бар. Layouts у Expo Router дозволяють винести спільні елементи інтерфейсу на рівень вище.

Основні функції Layout:

1. **Персистентність :** Елементи всередині Layout (наприклад, Tab Bar) не перемонтовуються при переході між дочірніми екранами. Це забезпечує плавність інтерфейсу.
2. **Абстракція навігації:** Ви визначаєте тип навігації (Stack, Tabs) один раз для цілої групи екранів.
3. **Глобальний стан:** Layout — ідеальне місце для обгортки екранів у `Providers` (Theme, Auth, QueryClient).

Архітектурна роль: Файл `_layout.tsx` автоматично стає батьківським для всіх файлів у своїй директорії.

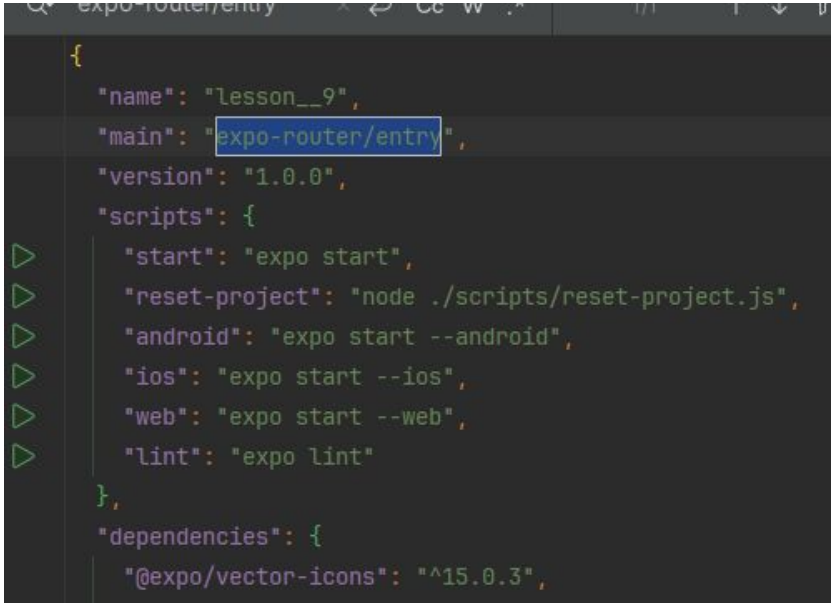
Механіка **Entry Point (expo-router/entry)**

На відміну від стандартного React Native, де точка входу — це `App.js`, Expo Router використовує спеціалізований модуль.

Роль `expo-router/entry`:

- **HMR (Hot Module Replacement):** Забезпечує коректне оновлення екранів без скидання стану навігаційного стека.
- **Error Handling:** Монтує глобальний `Error Boundary` для перехоплення критичних помилок рендеру на рівні всього додатка.

Це критичне налаштування, яке передає контроль над життєвим циклом додатка під управління роутера.



Convention over Configuration: Система правил

Для мінімізації коду Expo Router впроваджує систему зарезервованих назв, які мають пріоритет над звичайними файлами.

Матриця пріоритетів іменування:

1. **`index.tsx`**: Має найвищий пріоритет для поточного сегмента шляху.
2. **`_layout.tsx`**: Завжди виконується першим перед рендером дочірніх маршрутів.
3. **`+not-found.tsx`**: Спеціалізований роут для обробки неіснуючих адрес (аналог 404).
4. **`[id].tsx`**: Динамічний сегмент, що перехоплює будь-яке значення, якщо не знайдено точного збігу зі статичним файлом.

Index Routes — Вхідні точки сегментів

Файл `index.tsx` (або `.js`, `.ts`) є зарезервованим ім'ям, що позначає корінь будь-якого сегмента шляху. Це дозволяє створювати "чисті" URL без вказування назви файлу.

Механізм резолвінгу:

- `app/index.tsx` — відповідає маршруту `/`.
- `app/settings/index.tsx` — відповідає маршруту `/settings`.

Архітектурна перевага: Такий підхід дозволяє легко масштабувати окрему сторінку до цілого розділу. Якщо ваша сторінка `/profile` потребує додаткових вкладених екранів, ви просто перетворюєте `profile.tsx` на папку `profile/` з файлом `index.tsx` всередині, не змінюючи зовнішні посилання.

Компонент **<Link>:** Декларативна навігація

Компонент `<Link>` є основним інструментом для здійснення переходів між екранами. Він оптимізований для роботи на всіх платформах і автоматично обробляє натискання (onPress) на мобільних пристроях та атрибут `href` у Web.

Основні властивості:

- **`href`**: Шлях до цільового екрана (рядок або об'єкт).
- **Доступність**: Автоматично підтримує Accessibility-ролі для екранних дикторів.
- **Продуктивність**: Використовує нативні методи переходу без зайвих перерендерів.

Приклад використання:

```
import { Link } from 'expo-router';

<Link href="/settings">Перейти до налаштувань</Link>
```

Компонент **<Link>**: Декларативна навігація

```tsx
import { View, Text, StyleSheet } from 'react-native';
import { Link, Stack } from 'expo-router';

no usages
export default function HomeScreen() {
    return (
        <View style={styles.container}>
            <Stack.Screen options={{ title: 'Головна' }} />
            <Text style={styles.title}>Вітаємо у додатку!</Text>

            <Link href="/settings" style={styles.button}>
                ⚙️ Перейти в налаштування
            </Link>
        </View>
    );
}
```

```tsx
import { View, Text, StyleSheet } from 'react-native';
import {Link, Stack} from 'expo-router';

Show usages  new *
export default function SettingsIndex() {
    return (
        <View style={styles.container}>
            <Stack.Screen options={{ title: 'Налаштування' }} />
            <Text style={styles.title}>Налаштування</Text>

            <Link href="/" style={styles.link}>
                🏠 Повернутися на головну
            </Link>
        </View>
    );
}
```

Проп **asChild** та композиція компонентів

За замовчуванням `<Link>` огортає текст у нативний компонент натискання. Проте часто виникає потреба зробити «лінком» кастомну кнопку, іконку або складний UI-блок. Для цього використовується проп `asChild`.

Механізм роботи:

- `asChild` змушує `<Link>` передати всі свої властивості (включаючи обробник натискання) безпосередньо першому дочірньому елементу.

```
<Link href="/settings" asChild>
    <Pressable style={styles.customButton}>
        <Text style={styles.buttonText}>Налаштування ⚙</Text>
    </Pressable>
</Link>
```

Навігація за межі додатка (External Links)

Компонент `<Link>` універсальний: він розрізняє внутрішні маршрути додатка та зовнішні вебресурси.

Механізм External Linking:

- Якщо `href` починається з `http://` або `https://`, Expo Router використовує нативний модуль `Linking` (або `Web Browser` для Expo).
- На мобільних пристроях це відкриває браузер за замовчуванням.

```
<Link href="https://google.com">
  Відкрити документацію
</Link>
```

Dynamic Routes: Робота з параметрами **[id]**

Динамічна маршрутизація дозволяє використовувати один і той самий шаблон екрана для відображення різних даних на основі ідентифікатора в URL.

Синтаксис: Використання квадратних дужок у назві файлу: `app/post/[id].tsx`.

Механіка роботи:

- Будь-який запит типу `/post/123` або `/post/hello-world` буде перехоплений цим файлом.
- Значення 123 або `hello-world` буде доступне всередині компонента як параметр `id`.

Кейси використання:

- Картки товарів (`/product/[sku]`).
- Профілі користувачів (`/user/[username]`).
- Чат-кімнати (`/messages/[chatId]`).

```tsx
export default function ShopScreen() {
    const products = [
        { id: '101', name: 'iPhone 15' },
        { id: '102', name: 'MacBook Air' },
        { id: 'apple-watch', name: 'Apple Watch' }, // id може бути і рядком
    ];

    return (
        <View style={styles.container}>
            <Text style={styles.header}>Наші товари:</Text>
            {products.map((item) => (
                <Link key={item.id} href={`/products/${item.id}`} asChild>
                    <Pressable style={styles.item}>
                        <Text style={styles.itemText}>{item.name}</Text>
                    </Pressable>
                </Link>
            ))}
        </View>
    );
}
```

```tsx
import { View, Text, StyleSheet } from 'react-native';
import { useLocalSearchParams, Stack } from 'expo-router';

no usages  new *
export default function ProductDetails() {
    const { id } = useLocalSearchParams();

    return (
        <View style={styles.container}>
            {/* Динамічний заголовок хедера */}
            <Stack.Screen options={{ title: `Товар ID: ${id}` }} />

            <Text style={styles.title}>Деталі товару</Text>
            <Text style={styles.info}>Зараз ви переглядаєте товар з ідентифікатором: </Text>
            <Text style={styles.idBadge}>{id}</Text>
        </View>
    );
}
```

Catch-all Routes: Обробка нерегулярних шляхів

Якщо необхідно перехопити всі вкладені маршрути після певного сегмента, використовуються "загальні" маршрути зі синтаксисом розпакування (`...`).

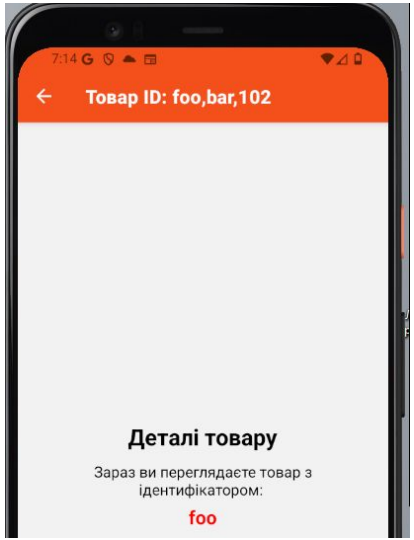
Синтаксис: `app/blog/[...slug].tsx`

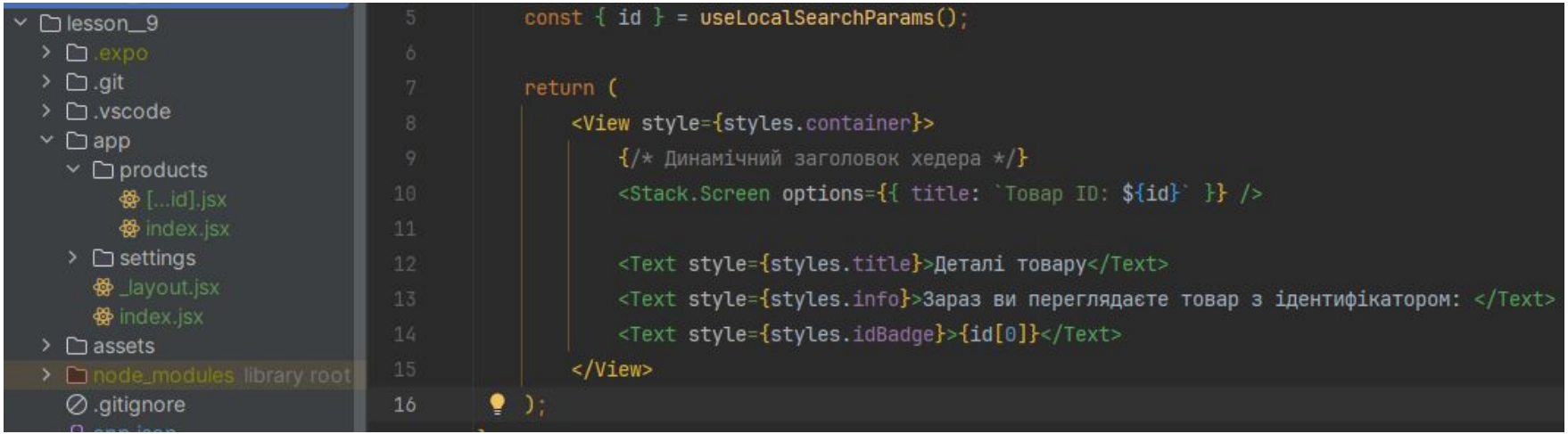
Відмінність від звичайного [id]:

- `[id]` перехоплює лише один сегмент (наприклад, `/blog/1`).
- `[...slug]` перехоплює будь-яку кількість сегментів (наприклад, `/blog/2024/05/12/my-post`).

Результат: Параметр `slug` буде переданий у компонент не як рядок, а як **масив** сегментів: `['2024', '05', '12', 'my-post']`. Це критично для побудови складних ієрархічних систем, таких як файлові менеджери.

```jsx
<View style={styles.container}>
    <Text style={styles.header}>Наші товари:</Text>
    {products.map((item) => (
        <Link key={item.id} href={`/products/foo/bar/${item.id}`} asChild>
            <Pressable style={styles.item}>
                <Text style={styles.itemText}>{item.name}</Text>
            </Pressable>
        </Link>
    ))}
</View>
```





Пріоритетність маршрутів

Коли декілька файлів можуть відповідати одному URL, Expo Router використовує алгоритм специфічності (аналогічно CSS).

Порядок пріоритету (від вищого до нижчого):

Static : Якщо є файл `app/about.tsx`, він відкриється завжди при переході на `/about`.

Dynamic: Якщо точного збігу немає, спрацює файл у дужках `app/[id].tsx`. Сюди потрапить все, що завгодно (наприклад, `/123` або `/user1`).

Catch-all: Файл `app/[...rest].tsx` — перехоплює все, що не підійшло іншим (наприклад, довгі шляхи `/news/2024/05/today`). Зазвичай це використовують для сторінки "404 Not Found".

Пріоритетність маршрутів

Приклад конфлікту: Якщо є файли `app/user/me.tsx` та `app/user/[id].tsx`:

- При переході на `/user/me` відкриється перший файл.
- При переході на `/user/123` відкриється другий.

Це дозволяє створювати специфічні сторінки (наприклад, "Мій профіль") окремо від загальних шаблонів профілів інших користувачів.

Керування історією: проп **replace**

Навігація за замовчуванням працює за принципом **Push** (додавання нового екрана в стек). Проте в певних сценаріях (авторизація, завершення замовлення) необхідно замінити поточний екран без можливості повернення назад.

Різниця між Push та Replace:

1. **Push (Default)**: Додає новий запит в історію. Користувач може натиснути кнопку «Назад».
2. **Replace**: Замінює поточний запис у стеку. Попередній екран видаляється з історії переходів.

Використання в коді:

```
<Link href="/products" replace asChild>
    <Pressable style={styles.customButton}>
        <Text style={styles.buttonText}>Продукти ⚙</Text>
    </Pressable>
</Link>
```

Хук **useRouter:** Програмний інтерфейс навігації

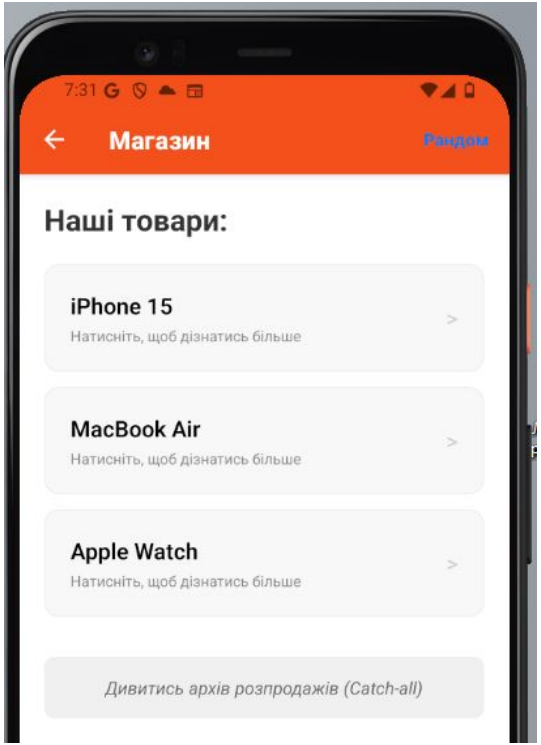
У випадках, коли перехід повинен відбутися не за кліком, а внаслідок певної події (після запиту до API, за таймером тощо), використовується хук `useRouter`.

Методи об'єкта router:

- **`push(path)`**: Додає маршрут до стека.
- **`replace(path)`**: Замінює поточний маршрут.
- **`back()`**: Повертає на попередній екран.
- **`canGoBack()`**: Перевіряє, чи є куди повертатися (корисно для кастомних кнопок "Назад").
- **`setParams(params)`**: Оновлює параметри поточного маршруту без переходу.

Приклад:

```tsx
import { Link, Stack, useRouter } from 'expo-router';

Show usages  new *
export default function ShopScreen() {
    const router = useRouter(); // Хук для програмної навігації

    const products = [
        { id: '101', name: 'iPhone 15' },
        { id: '102', name: 'MacBook Air' },
        { id: 'apple-watch', name: 'Apple Watch' },
    ];

    // Функція для переходу на випадковий товар
    Show usages  new *
    const goToRandomProduct = () => {
        const randomProduct = products[Math.floor(Math.random() * products.length)];
        router.push(`/products/${randomProduct.id}`);
    };
```



Передача параметрів через об'єкт **href**

Хоча `href` приймає рядок, для передачі великої кількості параметрів або складних шляхів зручніше використовувати об'єктний синтаксис. Це робить код більш читабельним і запобігає помилкам ручного формування рядка.

Структура об'єкта:

- **`pathname`**: Шлях до файлу або сегмент маршруту.
- **`params`**: Об'єкт з ключами та значеннями, які стануть частиною URL.

```
{products.map((item) => (
    <Link
        key={item.id}
        href={{
            pathname: "/products/[id]",
            params: { id: item.id, name: item.name, price: item.price }
        }}
        asChild
    >
        <Pressable style={styles.item}>
            <View>
                <Text style={styles.itemText}>{item.name}</Text>
                <Text style={styles.itemPrice}>{item.price}</Text>
            </View>
            <Text style={styles.arrow}>Деталі &gt;</Text>
        </Pressable>
    </Link>
))}
```

Хук **useLocalSearchParams:** Робота з контекстом екрана

Для отримання параметрів, переданих через динамічні сегменти (наприклад, `[id].tsx`), використовується хук `useLocalSearchParams`. Він повертає об'єкт, де ключі відповідають назвам файлів у квадратних дужках.

Особливості:

- **Scope**: Повертає параметри, релевантні тільки для **поточного** екрана.
- **Reactivity**: Значення автоматично оновлюються при зміні URL (наприклад, при переході з `/product/1` на `/product/2`).

```
const { id } = useLocalSearchParams();

return (
    <View style={styles.container}>
        {/* Динамічний заголовок хедера */}
        <Stack.Screen options={{ title: `Товар ID: ${id}` }} />

        <Text style={styles.title}>Деталі товару</Text>
        <Text style={styles.info}>Зараз ви переглядаєте товар з ідентифікатором: </Text>
        <Text style={styles.idBadge}>{id}</Text>
    </View>
);
```

useGlobalSearchParams vs useLocalSearchParams

Розуміння різниці між цими хуками є критичним для проектування складних вкладених інтерфейсів.

1. **`useLocalSearchParams`**:
   - Відображає параметри, специфічні для конкретного маршруту.
   - Використовується безпосередньо в екрані-компоненті.
2. **`useGlobalSearchParams`**:
   - Надає доступ до всіх параметрів у поточному навігаційному стані, включаючи параметри батьківських або сусідніх роутів.
   - **Кейс використання**: Коли в `_layout.tsx` (хедері або таб-барі) потрібно відобразити дані, які належать активному дочірньому екрану.

```tsx
import {Stack, useGlobalSearchParams} from "expo-router";
import {View} from "react-native";

no usages  new *
export default function RootLayout() {
    const params = useGlobalSearchParams();

    console.log(params)
    return (
        <View style={{flex: 1}}>
            <Stack
                screenOptions={{
                    headerStyle: {backgroundColor: '#f4511e'},
                    headerTintColor: '#fff',
                    headerTitleStyle: {fontWeight: 'bold'},
                }}
            />
        </View>
    );
}
```

Platform-Specific Routes

Expo Router дозволяє створювати різні маршрути для різних платформ, використовуючи розширення файлів.

- `index.ios.tsx` — відкриється тільки на iPhone.
- `index.android.tsx` — тільки на Android.
- `index.web.tsx` — тільки в браузері.

  Це дозволяє писати специфічний UX для кожної платформи, не використовуючи перевірок `Platform.OS === 'ios'`.

Компонент **<Slot />**: Механізм впровадження контенту

`Slot` — це спеціальний компонент, який виконує роль **динамічної області відображення** у файлі макета (`_layout.jsx`). Він визначає місце, куди буде підставлено вміст поточного маршруту.

Принцип роботи

- **Роль макета (`_layout.jsx`).** Використовується для визначення спільних елементів інтерфейсу:
  - Header
  - Footer
  - навігаційні панелі

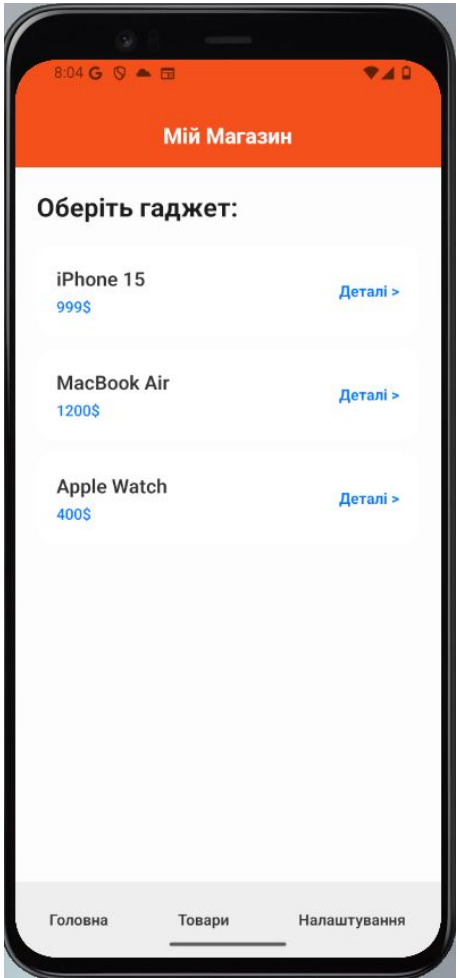
  Ці елементи залишаються незмінними при переході між сторінками.

- **Точка рендерингу (`<Slot />`).** Компонент `Slot` виступає як **контейнер**, у який автоматично підставляється:

  - відповідний екран (`index.jsx`, `[id].jsx` тощо) залежно від активного маршруту

- **Динамічна підстановка.** Вміст `Slot` змінюється відповідно до URL/шляху, без необхідності ручного керування навігацією.

```tsx
export default function RootLayout() {
    const { name } = useGlobalSearchParams();

    return (
        <View style={styles.container}>
            <View style={styles.header}>
                <Text style={styles.headerTitle}>
                    {name ? `👋 ${name}` : "Мій Магазин"}
                </Text>
            </View>

            <View style={styles.mainContent}>
                <Slot />
            </View>

            <View style={styles.footer}>
                <Link href="/" style={styles.navLink}>Головна</Link>
                <Link href="/products" style={styles.navLink}>Товари</Link>
                <Link href="/settings" style={styles.navLink}>Налаштування</Link>
            </View>
        </View>
    );
}
```



Типи навігаторів

Stack Layout: Нативна стекова навігація

Стекова навігація — це найбільш розповсюджений патерн у мобільних додатках. Expo Router використовує `Native Stack`, що забезпечує найвищу продуктивність.

Характеристики:

- **Нативні анімації:** Використовує системні переходи iOS та Android.
- **Header Management:** Автоматично генерує заголовок з кнопкою "Назад".
- **Оптимізація:** Екрани, що знаходяться нижче у стеку, не споживають ресурси CPU для рендерингу.

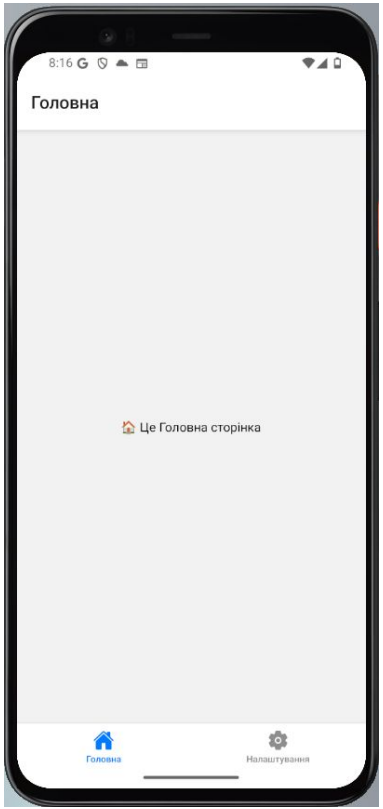
```tsx
import {Stack} from "expo-router";
import {View} from "react-native";

no usages  new *
export default function RootLayout() {
    return (
        <View style={{flex: 1}}>
            <Stack
                screenOptions={{
                    headerStyle: {backgroundColor: '#f4511e'},
                    headerTintColor: '#fff',
                    headerTitleStyle: {fontWeight: 'bold'},
                }}
            />
        </View>
    );
}
```

Tabs Layout: Проектування **Bottom Navigation**

Нижня панель навігації (Tabs) є стандартом для головних розділів додатка. Expo Router дозволяє налаштувати таби декларативно через спеціальний компонент.

Ключові особливості:

1. **Збереження стану:** При перемиканні між табами стан кожного екрана зберігається.
2. **Нативна поведінка:** Підтримка жестів та коректна висота панелі з урахуванням Safe Area.
3. **Icon Management:** Зручне налаштування іконок через `screenOptions`.




```jsx
import { Tabs } from 'expo-router';
import { Ionicons } from '@expo/vector-icons'; // Імпортуємо іконки

no usages  new *
export default function Layout() {
    return (
        <Tabs screenOptions={{ tabBarActiveTintColor: '#007AFF' }}>
            <Tabs.Screen
                name="index"
                options={{
                    title: 'Головна',
                    tabBarIcon: ({ color }) => <Ionicons name="home" size={24} color={color} />
                }}
            />

            <Tabs.Screen
                name="settings"
                options={{
                    title: 'Налаштування',
                    tabBarIcon: ({ color }) => <Ionicons name="settings" size={24} color={color} />
                }}
            />
        </Tabs>
    );
}
```

Drawer Layout: Бокове меню

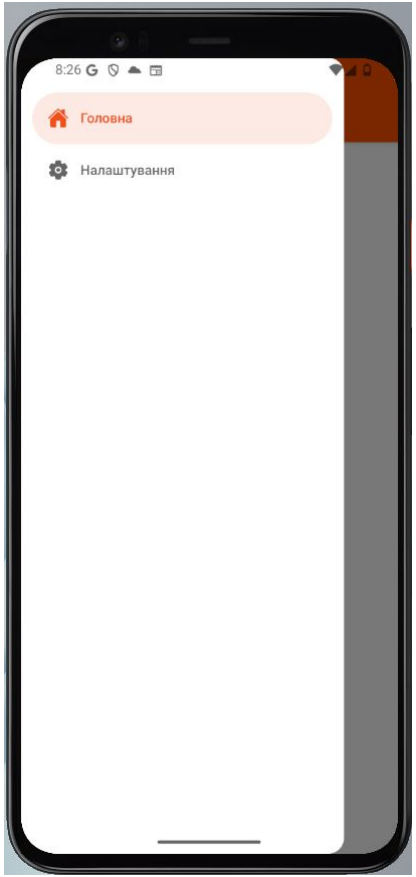
Drawer (бокова панель) часто використовується у великих додатках для доступу до другорядних розділів або налаштувань профілю.

Специфіка реалізації:

- Для використання Drawer в Expo Router зазвичай потрібна додаткова бібліотека `@react-navigation/drawer`.
- У Expo Router він доступний через експорт `Drawer` з `expo-router/drawer`.

Функціонал:

- Підтримка жесту "swipe" для відкриття.
- Налаштування ширини панелі, кольору оверлея та типу анімації (slide vs front).



```jsx
import { GestureHandlerRootView } from 'react-native-gesture-handler';
import { Drawer } from 'expo-router/drawer';
import { Ionicons } from '@expo/vector-icons';

no usages  new *
export default function Layout() {
    return (
        <GestureHandlerRootView style={{ flex: 1 }}>
            <Drawer screenOptions={{
                drawerActiveTintColor: '#f4511e',
                headerStyle: { backgroundColor: '#f4511e' },
                headerTintColor: '#fff'
            }}>
                <Drawer.Screen
                    name="index"
                    options={{
                        drawerLabel: 'Головна',
                        title: 'Головний екран',
                        drawerIcon: ({ color }) => <Ionicons name="home" size={22} color={color} />,
                    }}
                />
                <Drawer.Screen
                    name="settings"
                    options={{
                        drawerLabel: 'Налаштування',
                        title: 'Налаштування профілю',
                        drawerIcon: ({ color }) => <Ionicons name="settings" size={22} color={color} />,
                    }}
                />

            </Drawer>
        </GestureHandlerRootView>
    );
```

Screen Options — Декларативне керування інтерфейсом

Кожен навігатор (Stack, Tabs, Drawer) дозволяє конфігурувати зовнішній вигляд екранів через проп `screenOptions` (глобально) та `options` (індивідуально).

Рівні конфігурації:

1. **Глобальний (`screenOptions`)**: Визначає спільний стиль для всіх екранів у межах даного Layout (наприклад, колір фону хедера).
2. **Індивідуальний (`options`)**: Перезаписує глобальні налаштування для конкретного маршруту.

```tsx
import { GestureHandlerRootView } from 'react-native-gesture-handler';
import { Drawer } from 'expo-router/drawer';

no usages  new *
export default function RootLayout() {
    return (
        <GestureHandlerRootView style={{ flex: 1 }}>
            <Drawer
                screenOptions={{
                    // Глобальні налаштування інтерфейсу
                    headerStyle: { backgroundColor: '#f4511e' },
                    headerTintColor: '#fff',
                    headerTitleStyle: { fontWeight: 'bold' },
                    headerTitle: "Додаток",
                }}
            >
                <Drawer.Screen name="index" options={{ drawerLabel: 'Головна' }} />
                <Drawer.Screen name="settings" options={{ drawerLabel: 'Налаштування' }} />
            </Drawer>
        </GestureHandlerRootView>
    );
}
```

Nested Layouts — Архітектура складних додатків

Додатки рідко використовують лише один тип навігації. Типовий патерн — це **Tabs**, де кожен таб є окремим **Stack**.

Як реалізувати вкладеність:

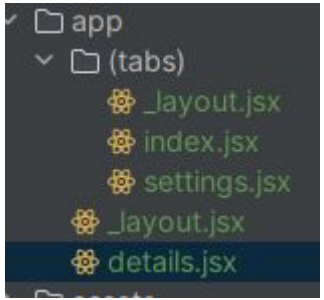
1. У корені (`app/_layout.tsx`) визначається основний `Stack`.
2. Створюється папка для головного екрана, наприклад `app/(tabs)/_layout.tsx`, де визначається `Tabs`.
3. Expo Router автоматично об'єднує ці структури.

Groups

Групи маршрутів — це спосіб організувати файли в папці `app` так, щоб вони мали спільний Layout, але при цьому їхня назва **не відображалася в URL-адресі**.

Ключові особливості:

1. **Ігнорування в URL**: Якщо папка називається `(tabs)`, то шлях до файлу `app/(tabs)/index.jsx` буде просто `/`, а не `/tabs/`.
2. **Спільний Layout**: Всі файли всередині `(tabs)` підпорядковуються файлу `app/(tabs)/_layout.jsx`.
3. **Організація коду**: Дозволяє відокремити логіку (наприклад, окремо таби, окремо авторизацію `(auth)`) без зміни структури посилань.




```jsx
import { Stack } from "expo-router";

no usages  new *
export default function RootLayout() {
    return (
        <Stack>
            {/* Реєструємо групу табів. Назва папки (tabs) в URL не потрапить */}
            <Stack.Screen name="(tabs)" options={{ headerShown: false }} />
            <Stack.Screen name="details" options={{ title: 'Деталі продукту' }} />

        </Stack>
    );
}
```

```jsx
import { Tabs } from "expo-router";
import { Ionicons } from '@expo/vector-icons';

no usages  new *
export default function TabLayout() {
    return (
        <Tabs screenOptions={{ tabBarActiveTintColor: 'blue' }}>
            <Tabs.Screen
                name="index"
                options={{
                    title: 'Головна',
                    tabBarIcon: ({ color }) => <Ionicons name="home" size={24} color={color} />
                }}
            />
            <Tabs.Screen
                name="settings"
                options={{
                    title: 'Налаштування',
                    tabBarIcon: ({ color }) => <Ionicons name="settings" size={24} color={color} />
                }}
            />
        </Tabs>
    );
}
```

Умовний рендер та **Protected Routes**

Декларативний підхід Expo Router дозволяє легко реалізувати захищені маршрути через компонент `<Redirect />`.

Алгоритм роботи:

1. У Layout перевіряється стан авторизації (наприклад, наявність токена).
2. Якщо користувач не авторизований, він автоматично перенаправляється на сторінку `/login`.

Це гарантує, що неавторизований користувач ні за яких умов захищений контент, навіть якщо спробує перейти за прямим посиланням.

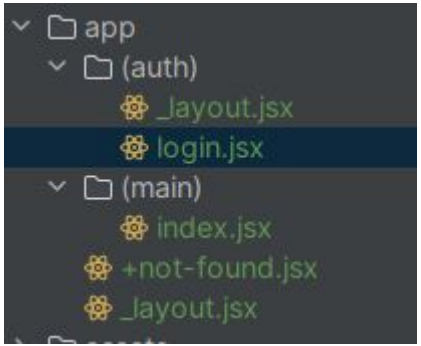
```tsx
import { Stack, Redirect, useSegments } from "expo-router";

no usages  new *
export default function RootLayout() {
    const isLoggedIn = false;
    const segments = useSegments();

    const isOnLoginPage = segments[0] === "login";

    if (!isLoggedIn && !isOnLoginPage) {
        return <Redirect href="/login" />;
    }

    return <Stack />;
}
```

Login Flow — Приклад архітектури **(auth)**

Розглянемо класичний сценарій: вхід у додаток.

Структура папок:

1. `app/(auth)/login.tsx`
2. `app/(auth)/_layout.tsx` (тут Stack з прихованим хедером).



Логіка переходу: Після успішного запиту до API ми виконуємо `router.replace('/(main)')`. Використання `replace` замість `push` є критичним, щоб користувач не міг повернутися назад на сторінку логіна за допомогою кнопки "Назад".

```jsx
import { View, Text, Button, StyleSheet } from 'react-native';
import { useRouter } from 'expo-router';

no usages  new *
export default function LoginScreen() {
    const router = useRouter();

    Show usages  new *
    const handleLogin = () => {
        router.replace('/(main)');
    };

    return (
        <View style={styles.container}>
            <Text style={styles.title}>Вхід у додаток</Text>
            <Button title="УВІЙТИ" onPress={handleLogin} />
        </View>
    );
}
```

Додаткові можливості

Аналіз навігаційного графа

При використанні груп `(auth)`, `(tabs)` та вкладених стеків важливо розуміти, як виглядає фінальний граф навігації.

Інструменти налагодження:

- **Expo Router Sitemap**: У режимі розробки перейти за маршрутом `/_sitemap`, щоб побачити всі зареєстровані шляхи.
- **React Navigation DevTools**: Дозволяє бачити поточний стан стека та параметри кожного екрана в реальному часі.

```
<Link href="/_sitemap">
    <Text>Open Sitemap</Text>
</Link>
```

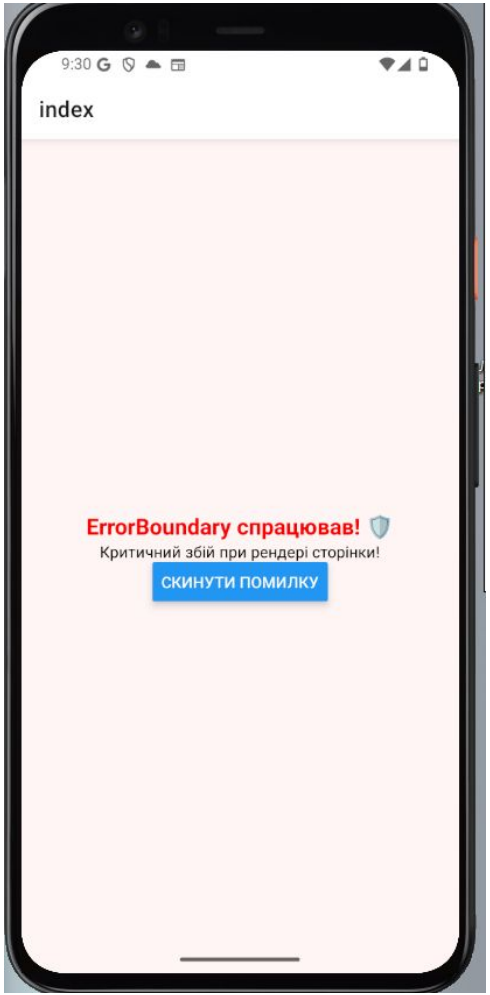
Обробка помилок через **Error Boundaries**

В Expo Router кожен сегмент маршруту може мати власну стратегію обробки критичних помилок . Це реалізується через спеціальну назву файлу або експорт `ErrorBoundary`.

Механізм ізоляції:

- Якщо помилка стається всередині конкретного екрана, Expo Router шукає найближчий `ErrorBoundary` в ієрархії папок.
- Це дозволяє зламатись лише одній частині додатка (наприклад, тільки стрічці новин), залишаючи навігацію та інші таби робочими.

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

no usages  new *
export function ErrorBoundary({ error, retry }) {
    return (
        <View style={styles.errorContainer}>
            <Text style={styles.errorTitle}>ErrorBoundary спрацював! 🛡️</Text>
            <Text>{error.message}</Text>
            <Button title="Скинути помилку" onPress={retry} />
        </View>
    );
}

no usages  new *
export default function HomeScreen() {
    const [shouldCrash, setShouldCrash] = useState(false);

    if (shouldCrash) {
        throw new Error("Критичний збій при рендері сторінки!");
    }

    return (
        <View style={styles.container}>
            <Text style={styles.text}>Статус: Все ок ✅</Text>
            <Button
                title="ЗЛАМАТИ"
                color="red"
                onPress={() => setShouldCrash(true)}
            />
        </View>
    );
}
```

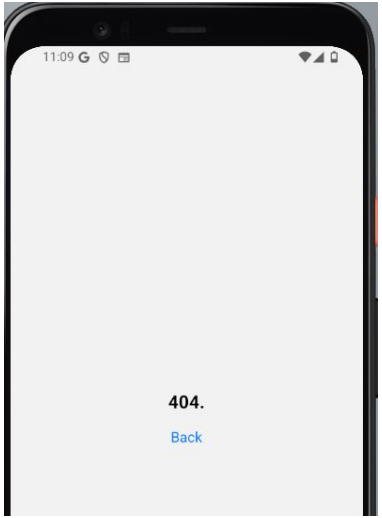


Екран **404** — Файл **+not-found.tsx**

Коли користувач переходить за неіснуючим посиланням (через Deep Link або помилку в коді), Expo Router шукає файл `+not-found.tsx`.

Особливості:

1. **Синтаксис:** Символ + вказує на те, що це системний маршрут.
2. **Глобальність:** Зазвичай такий файл знаходиться у корені `/app`, щоб він перехоплював усі помилкові шляхи в додатку.
3. **Web-parity:** У браузері цей екран повертає коректний HTTP статус 404 для пошукових роботів.



Safe Area в Layout — Інтеграція SafeAreaProvider

При використанні кастомних Layouts важливо враховувати "небезпечні зони" (вирізи, індикатори жестів).

Правильна ініціалізація: `SafeAreaProvider` має бути у `app/_layout.tsx`. Всі дочірні Layouts повинні використовувати `useSafeAreaInsets` або компонент `SafeAreaView`, для запобігання перекриття контенту системними елементами керування

Themes - Інтеграція з **React Navigation Themes**

Expo Router підтримує системні теми (Light/Dark), що дозволяє додатку автоматично змінювати колірну схему відповідно до налаштувань ОС.

Механізм роботи:

1. Використовуйте `ThemeProvider` з `@react-navigation/native`.
2. Передайте об'єкт `DefaultTheme` або `DarkTheme`.
3. Expo Router автоматично оновить кольори хедера, таб-бару та фону екранів.

Архітектурна порада: Найкраще місце для `ThemeProvider` — це кореневий `app/_layout.tsx`. Це гарантує, що тема буде доступна кожному екрану через контекст.

```tsx
import { Stack } from "expo-router";
import { ThemeProvider, DarkTheme, DefaultTheme } from "@react-navigation/native";
import { useColorScheme } from "react-native";

no usages  new *
export default function RootLayout() {
    // Отримуємо системну тему: 'light' або 'dark'
    const colorScheme = useColorScheme();

    return (
        // Вибираємо об'єкт теми залежно від налаштувань ОС
        <ThemeProvider value={colorScheme === 'dark' ? DarkTheme : DefaultTheme}>
            <Stack>
                <Stack.Screen name="index" options={{ title: "Головна" }} />
            </Stack>
        </ThemeProvider>
    );
}
```

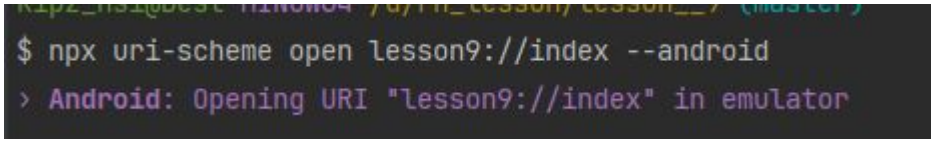
Deep Linking за замовчуванням - Кожен файл це **URL**

В звичайному React Native налаштування посилань (`myapp://page`) вимагає ручного мапінгу кожного екрана [Deep linking | React Navigation]. В Expo Router ця проблема вирішена на рівні архітектури.

- Якщо у вас є файл `app/index.jsx`, він **автоматично** стає доступним за посиланням `scheme://index`.
- Роутер сам розбирає URL, витягує параметри та монтує потрібний стек екранів.

Переваги:

- Немає дублювання логіки між структурою папок та конфігом лінкінгу.
- Зменшення кількості багів при переході з push-повідомлень на конкретний екран.



Scheme vs Universal Links — Рівні інтеграції з ОС

Існує два основні способи, як операційна система розуміє, що посилання має відкрити саме ваш додаток.

1. **Custom URL Schemes (`my-app://`)**:
   - Найпростіший спосіб. Налаштовується в `app.json` через поле `scheme`.
   - Мінус: Інший додаток може зареєструвати таку саму схему.
2. **Universal Links (iOS) / App Links (Android)**:
   - Використовують стандартні HTTP-посилання (наприклад, `https://my-app.com/user/42`).
   - Потребують підтвердження володіння доменом.
   - Якщо додаток не встановлено, посилання відкривається в браузері як звичайний сайт.

About App Links | App architecture | Android Developers

Expo Router on Web — Побудова SPA

Expo Router робить розробку для Web максимально наближеною до Next.js. При запуску `npx expo start --web` додаток перетворюється на SPA.

Особливості Web-версії:

- **History API**: Працюють кнопки "Вперед" та "Назад" у браузері.
- **SEO-friendly**: Кожна сторінка має свій URL, що дозволяє пошуковим ботам індексувати контент.
- **Fast Refresh**: Миттєве оновлення сторінки в браузері при зміні коду.

Deployment

Деплой веб-версії Expo Router відрізняється від мобільної збірки.

Static Export (`npx expo export --platform web`): Генерація набору HTML/JS файлів, які можна завантажити на будь-який хостинг (S3, GitHub Pages, Netlify).

```
$ npx expo export --platform web
React Compiler enabled
Starting Metro Bundler
Static rendering is enabled. Learn more: https://docs.expo.dev/router/reference/static-rendering/
Web node_modules\expo-router\entry.js 55.9% (368/493)
λ node_modules\expo-router\node\render.js 91.8% (569/594)
```

```
› web bundles (1):
_expo/static/js/web/entry-b8eb393e534e049805a7d09094b78040.js (992 kB)

› Static routes (4):
/ (index) (19.7 kB)
/modal (20.4 kB)
/_sitemap (17.7 kB)
/+not-found (17.7 kB)

Exported: dist
```

Файл **+html.tsx** — Кастомізація **Web**-версії

Для проектів, що працюють у Web, Expo Router дозволяє редагувати кореневий HTML-шаблон через файл `+html.tsx`. Це аналог `index.html` у React-додатках, але з підтримкою SSR (Server Side Rendering).

Можливості:

- Додавання зовнішніх скриптів (аналітика).
- Налаштування мета-тегів для SEO (Viewport, Favicons).

```tsx
import { ScrollViewStyleReset } from 'expo-router/html';
import { type PropsWithChildren } from 'react';

/**
 * Цей компонент рендериться лише на сервері (SSR).
 * Він не доступний у середовищі React Native на iOS/Android.
 */
no usages  new *
export default function RootHtml({ children }: PropsWithChildren) {
    return (
        <html lang="uk">
        <head>
            <meta charSet="utf-8" />
            <meta httpEquiv="X-UA-Compatible" content="IE=edge" />
            <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no" />

            {/* Скидання стилів скролу для коректної роботи на десктопі */}
            <ScrollViewStyleReset />

            {/* Кастомні мета-теги або стилі */}
            <style dangerouslySetInnerHTML={{ __html: `body { background-color: #fafafa; }` }} />
            <title>Test</title>
        </head>
        <body>{children}</body>
        </html>
    );
}
```

SEO та Мета-теги — Компонент <Head />

Для того, щоб посилання на додаток красиво виглядало в соцмережах (Open Graph) та індексувалося Google, необхідно керувати мета-даними.

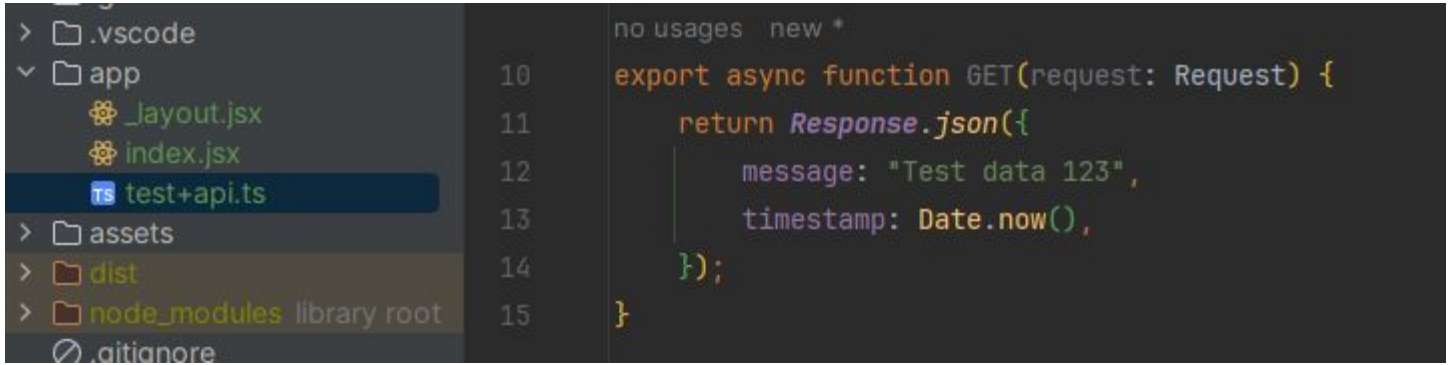
Використання `expo-router/head`:

```jsx
import { View, Text } from 'react-native';
import Head from 'expo-router/head';

no usages  new *
export default function HomeScreen() {
    return (
        <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center', gap: 20 }}>
            {/* Мета-теги для Web (ігноруються на мобільних пристроях) */}
            <Head>
                <title>Мій Додаток | Головна</title>
                <meta name="description" content="Це опис для Google та соцмереж" />
            </Head>

            <Text>Головний екран</Text>

        </View>
    );
}
```

API Routes (Server-side) — Файли +api.ts

Починаючи з SDK 50, Expo Router дозволяє створювати серверні обробники запитів всередині папки `app/`.

Будь-який файл із суфіксом `+api.ts` стає серверною функцією.



```
const [data, setData] = useState(null);
const [loading, setLoading] = useState(false);

Show usages  new *
const fetchData = async () => {
    setLoading(true);
    try {
        // Маршрут відповідає назві файлу (test+api.ts -> /test)
        const response = await fetch('/test');
        const json = await response.json();
        setData(json);
    } catch (error) {
        console.error("API Error:", error);
    } finally {
        setLoading(false);
    }
};
```

Handling Requests: GET, POST, PUT

API Routes підтримують стандартні HTTP-методи.

- **GET**: Отримання даних.
- **POST**: Створення записів.
- **PUT/PATCH**: Оновлення даних.
- **DELETE**: Видалення.

Перезапис шляхів

- **Redirects**: Автоматичне перенаправлення (наприклад, зі старого `/about-us` на новий `/about`).
- **Rewrites**: Дозволяє відображати контент одного файлу за зовсім іншою адресою, не змінюючи структуру папок. Це критично для SEO та підтримки старих Deep Links.

Redirects and rewrites - Expo Documentation