Розробка мобільних додатків

Лекція 7 - Файлова система

Файлова система мобільних пристроїв

Файлова система мобільного пристрою - це системний механізм ОС, який відповідає за організацію, зберігання, іменування та цілісність даних (файлів, директорій, кешу) на фізичних носіях пристрою.

Специфіка мобільних платформ:

- **Ізоляція:** Кожен додаток працює у власному ізольованому середовищі для запобігання несанкціонованому доступу до даних інших програм.
- **Жорстке квотування:** Мобільні ОС агресивно керують пам'яттю і можуть самостійно очищати певні директорії (наприклад, кеш) для забезпечення стабільності системи
- **Життєвий цикл додатка:** Операційна система може призупиняти або завершувати додатки у фоновому режимі, що впливає на роботу з файлами.
- **Оптимізація під енергоспоживання:** Операції читання/запису оптимізуються для зменшення використання батареї та ресурсів пристрою.

Основні зони файлової системи

Файлову систему поділяють на три основні зони:

Внутрішнє сховище (Internal Storage)

- Зберігання приватних даних додатку (бази даних, конфігурації, локальні профілі).
- Доступне тільки самому додатку. Ні користувач (без root-прав), ні інші додатки не мають доступу до цієї директорії).
- Дані безповоротно видаляються системою під час деінсталяції додатку.

Зовнішнє сховище (External Storage)

- Зберігання публічних або об'ємних даних (медіафайли, завантаження), які можуть використовуватися іншими програмами.
- Може бути вбудованим або на SD-карті.
- Потребує явного дозволу від користувача (`READ_EXTERNAL_STORAGE`, `WRITE_EXTERNAL_STORAGE`).

Кеш (Cache)

- Тимчасові файли. Зберігання даних для швидкого доступу (наприклад, завантажені мініатюри зображень).
- Операційна система залишає за собою право автоматично очистити цю папку при критичній нестачі вільного місця на пристрої.

Призначення та сценарії використання локального сховища

Типові сценарії збереження даних:

- **Офлайн-доступ (Offline First):** Завантаження та збереження статичного контенту (PDF-документи, відеофайли, JSON-дані) для забезпечення працездатності додатку без підключення до мережі Інтернет.
- **Кешування ресурсів:** Тимчасове збереження "важких" медіафайлів (зображення, відповіді API) з метою економії мобільного трафіку та прискорення рендерингу інтерфейсу.
- **Користувацький контент:** Збереження згенерованих даних, таких як нотатки, створені документи або прикріплені файли.
- **Проміжне збереження медіа:** Тимчасове збереження фотографій чи відео безпосередньо перед їх вивантаженням (публікацією) на віддалений сервер.
- **Системне логування:** Ведення локального журналу подій, фіксація критичних помилок та аналітики для подальшого діагностування роботи додатку.

Робота з файловою системою в **React Native**

React Native Core API **не має** вбудованих модулів для повноцінної взаємодії з локальною файловою системою пристрою.

В Expo стандартом є бібліотека **`expo-file-system`**, яка надає уніфікований інтерфейс для платформ Android IOS.

[FileSystem - Expo Documentation]()

Ключові можливості **expo-file-system**

- **Операції вводу/виводу:** Зчитування та запис файлів як у форматі текстових рядків, так і у вигляді бінарних даних (через кодування Base64).
- **Мережева взаємодія:** Можливість фонового або негайного завантаження файлів з Інтернету безпосередньо на фізичний диск пристрою.
- **Робота з метаданими:** Отримання системної інформації про об'єкти (розмір файлу, дата модифікації, MD5-хеш).
- **Керування структурою:** Створення, видалення директорій та перевірка наявності об'єктів за заданим шляхом.

expo-file-system

Expo FileSystem

A library that provides access to the local file system on the device.

Android · iOS · tvOS · Included in Expo Go

Ask AI · GitHub · npm · Changelog · Bundled version: ~55.0.10 · Copy page

Встановлення:

```
npx expo install expo-file-system
```

Імпорт:

```
import { File, Directory, Paths } from 'expo-file-system';
```

expo-file-system TS

55.0.10 • Public • Published 7 days ago

Expo File System

Provides access to the local file system on the device.

API documentation

- Documentation for the latest stable release
- Documentation for the main branch

Installation in managed Expo projects

For **managed** Expo projects, please follow the installation instructions in the **API documentation for the latest stable release**.

Installation in bare React Native projects

For bare React Native projects, you must ensure that you have **installed and configured the `expo` package** before continuing.

Install

```
> npm i expo-file-system
```

Repository

Homepage

Weekly Downloads

3 295 823

Version	License
55.0.10	MIT

app.json

Конфігурація в `app.json` додається лише у випадках, коли потрібні **нативні можливості платформи**, наприклад:

- `supportsOpeningDocumentsInPlace` - дозволяє відкривати документи без копіювання (iOS)
- `enableFileSharing` - дозволяє обмін файлами через Finder / iTunes (iOS)

Example app.json with config plugin

app.json

```
{
  "expo": {
    "plugins": [
      [
        "expo-file-system",
        {
          "supportsOpeningDocumentsInPlace": true,
          "enableFileSharing": true
        }
      ]
    ]
  }
}
```

https://docs.expo.dev/versions/latest/sdk/filesystem/#configurable-properties

Еволюція **expo-file-system**: Перехід до ООП

У нових версіях **`expo-file-system`** відбувся перехід від процедурного API до **об’єктно-орієнтованого підходу (OOP)**.
Раніше робота з файлами виконувалась через окремі функції модуля (`readAsStringAsync`, `writeAsStringAsync`, `deleteAsync`).

Тепер файли та директорії представлені **окремими об’єктами**, які інкапсулюють дані та операції над ними.

Expo FileSystem (legacy)

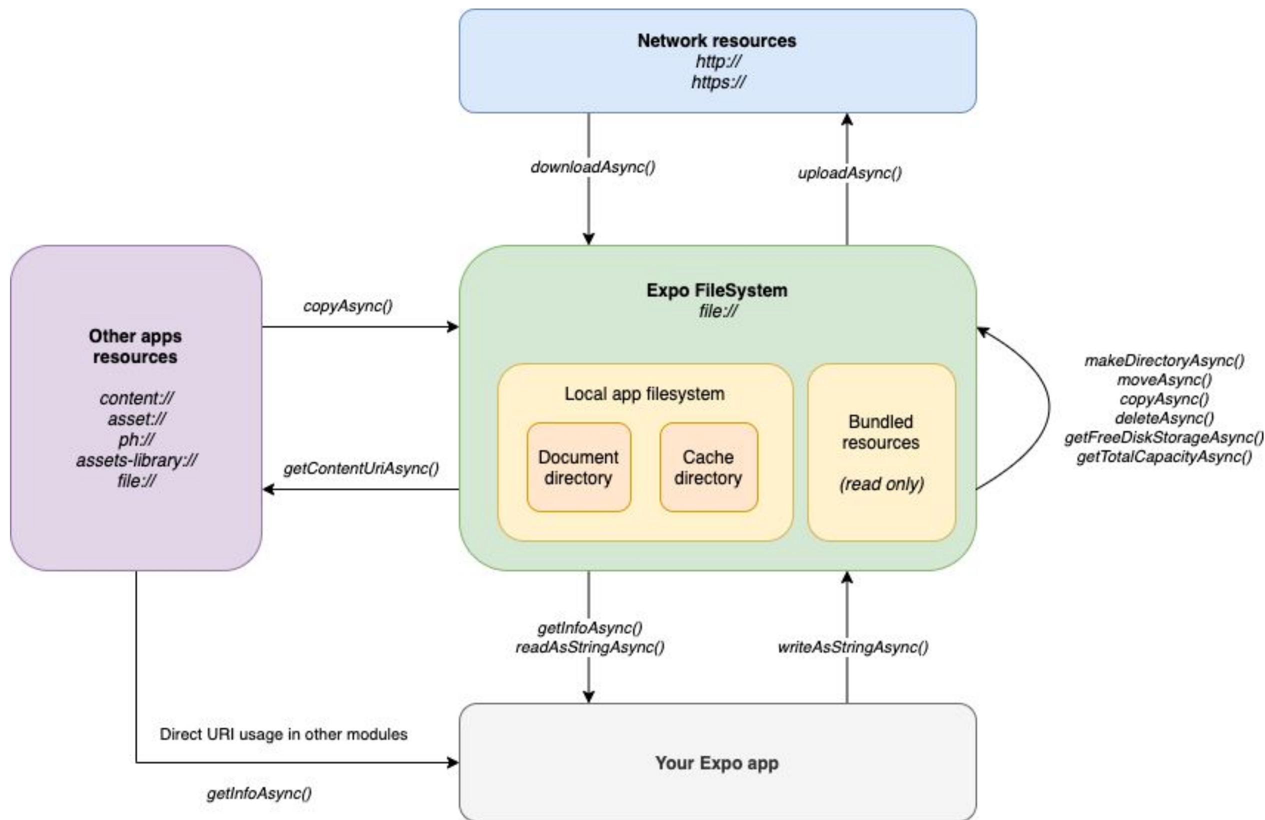
A library that provides access to the local file system on the device.

Android | iOS | tvOS | Included in Expo Go

Ask AI | GitHub | npm | Changelog | Bundled version: ~55.0.10 | Copy page

The `legacy` version of the FileSystem API is included in the `expo-file-system` library. It can be used alongside the modern API for backward compatibility reasons.

Схема роботи **Expo FileSystem (legacy)**



Permissions

Android

Android, потрібні дозволи на доступ до **зовнішньої файлової системи**. `READ_EXTERNAL_STORAGE` — читання файлів; `WRITE_EXTERNAL_STORAGE` — запис файлів; `INTERNET` — завантаження файлів з мережі

iOS

iOS використовує **sandbox-модель** — кожен додаток має власну ізольовану файлову систему. Тому **додаткові дозволи не потрібні**, якщо робота відбувається у внутрішніх директоріях додатку.

*Операційна система **Android історично використовувала модель спільного доступу до зовнішнього сховища**, де різні додатки могли працювати з файлами у спільних директоріях (наприклад, медіафайли, документи). Це архітектурне рішення сильно вплинуло і на **безпеку**, і на **модель дозволів** у двох системах.*

Схеми маршрутизації **(URI Schemes)** у мобільних ОС

Для однозначної ідентифікації та локалізації ресурсів мобільні операційні системи використовують уніфіковані ідентифікатори ресурсів (URI), які починаються з певних префіксів (схем).

Supported URI schemes

Expo робить **уніфікацію URI-схем між платформами**. На Android у системі одночасно існують `file://`, `content://`, `asset://`, а в iOS переважно `file://`. Expo перетворює це на одну зрозумілу модель, щоб розробник працював просто з **URI файлу**, не думаючи про внутрішню архітектуру ОС.

file://

Універсальний URI для вказання **шляху до локального файлу в системі**.

Приклад:

```
file:///storage/emulated/0/Download/myfile.txt
```

Використання:
Вказує на файл, який збережено напряму в файловій системі пристрою (зовнішнє або внутрішнє сховище).

content://

Це загальна схема URI, яка використовується для доступу до даних, що надаються **Content Provider-ами** в Android.

Приклади:

- Галерея фото
- Контакти
- Документи

Використання:
Android-додатки використовують `content://` для безпечного обміну файлами. Наприклад, вибраний файл через системний "file picker" матиме саме таку адресу.

asset://

Вказує на **вбудовані ресурси** (assets), які зберігаються в папці `assets/` вашого Android-проєкту.

Використання:
Можна використовувати для доступу до статичних файлів (зображення, HTML, JSON тощо), які були додані до проєкту ще під час розробки.

Ініціалізація об'єктів класу **Directory**

Клас `Directory` є фундаментальним будівельним блоком для організації файлової структури вашого додатка. Він представляє директорію на файловій системі.

Синтаксис ініціалізації: Конструктор класу приймає масив рядків або існуючих об'єктів `Directory`, які об'єднуються для створення кінцевого шляху (URI).

```
import { Directory, Paths } from 'expo-file-system';

// Створення посилання на підпапку "downloads" у кеші додатка
const downloadsFolder = new Directory(Paths.cache, 'downloads');

// Створення глибоко вкладеного шляху
const userAvatarsDir = new Directory(Paths.document, 'users', 'avatars');
```

```
LOG  Downloads URI: file:///data/user/0/host.exp.exponent/cache/downloads/
LOG  Avatars URI: file:///data/user/0/host.exp.exponent/files/users/avatars/
```

Виклик new `Directory()` є суто синхронною операцією в пам'яті JavaScript. Він формує URI (наприклад, `file:///data/user/0/.../cache/downloads/`), але **не здійснює** жодних звернень до фізичного диска і не створює папку автоматично.

Ключові властивості класу **Directory**

Після створення екземпляра `Directory`, розробник отримує доступ до синхронних властивостей, які описують цей об'єкт.

- **`exists` (boolean):** Повертає `true`, якщо директорія фізично існує на диску і додаток має права доступу до неї. Інакше — `false`.
- **`uri` (string):** Абсолютний шлях до директорії (Read-only). Автоматично гарантує правильний префікс (наприклад, `file://`).
- **name (string):** Кінцеве ім'я папки (наприклад, `"avatars"` з попереднього слайда).
- **`parentDirectory` (Directory):** Повертає новий екземпляр класу `Directory`, який вказує на батьківську папку. Дозволяє легко підніматися вгору по дереву файлової системи.

```
console.log("Downloads URI:", downloadsFolder);
console.log("Avatars URI:", avatarsDir.parentDirectory.uri );
```

Фізичне створення директорій на диску

Для того щоб папка, описана об'єктом `Directory`, фактично з'явилася на накопичувачі, необхідно викликати метод `.create()`. Метод приймає об'єкт параметрів, який дозволяє керувати поведінкою у конфліктних ситуаціях:

```
if (!userAvatarsDir.exists) {
  userAvatarsDir.create({
    intermediates: true, // Аналог команди 'mkdir -p'
    idempotent: true     // Запобігає помилці, якщо папка вже існує
  });
}
```

`intermediates: true`: Найважливіший параметр. Якщо ви намагаєтеся створити шлях `директорія1/директорія2/директорія3`, а `директорія1` ще не існує, система автоматично створить весь ланцюжок. Без цього параметра програма видасть фатальну помилку .

`idempotent: true`: Робить операцію безпечною для багаторазового виклику. Якщо папка вже створена, метод просто завершиться успіхом без викидання винятку.

```
LOG  Directory info: {"creationTime": 1772735511000, "exists": true, "files": [], "md5": null, "modificationTime": 1772735511824, "size": 0, "uri": "file:///data/user/0/host.exp.exponent/files/users/avatars"}
```

Читання вмісту директорії (Listing)

Щоб дізнатися, які файли та підпапки містяться у певній директорії, використовується метод `.list()`.

Особливості сучасного методу `list()`: На відміну від старого API, який повертав масив простих текстових імен (`['file.txt', 'photo.jpg']`), сучасний метод повертає масив готових об'єктів — екземплярів `File` та `Directory`.

```
// отримуємо список об'єктів
const items = documentsDir.list();

for (const item of items) {
  if (item instanceof File) {
    console.log("Файл:", item.name, "| URI:", item.uri);
  }
  if (item instanceof Directory) {
    console.log("Папка:", item.name, "| URI:", item.uri);
  }
}
```

```
// директорія документів додатка
const documentsDir = new Directory(Paths.document);
```

```
LOG  Файл: profileInstalled | URI: file:///data/user/0/host.exp.exponent/files/profileInstalled
LOG  Папка: phenotype_storage_info | URI: file:///data/user/0/host.exp.exponent/files/phenotype_storage_info/
LOG  Папка: ExperienceData | URI: file:///data/user/0/host.exp.exponent/files/ExperienceData/
```

Це спрощує подальшу взаємодію з елементами: розробнику не потрібно вручну склеювати базовий шлях з іменами файлів для їх читання.

Метадані та інтроспекція директорій

Для отримання детальної технічної інформації про директорію використовується метод `.info()`. Він корисний для побудови UI файлових менеджерів або для аналітики.

Метод повертає об'єкт `DirectoryInfo`:

- **`size` (number | null):** Загальний розмір директорії (включно з усіма вкладеними файлами) у байтах.
- **`creationTime` (number | null):** Час створення папки (у мілісекундах з епохи Unix). *Примітка: на старих версіях Android (до API 26) може повертати null через системні обмеження.*
- **`modificationTime` (number):** Час останньої зміни вмісту директорії.
- **`files` (string[]):** Опціонально повертає список імен файлів (для швидкого доступу без ініціалізації об'єктів через `.list()`).

```
// отримуємо технічну інформацію
const info = await usersDir.info();

console.log("URI:", usersDir.uri);
console.log("Exists:", info.exists);
console.log("Size:", info.size, "bytes");
console.log("Created:", info.creationTime);
console.log("Modified:", info.modificationTime);
```

Копіювання директорій (Deep Copy)

У процесі роботи додатку виникає потреба зробити резервну копію конфігурацій або кешувати цілу структуру папок. Для цього використовується метод `.copy()`.

```
const sourceDir = new Directory(Paths.document, 'user_data');
const backupDir = new Directory(Paths.cache, 'user_data_backup');

// Створення повної копії папки та всього її вмісту
sourceDir.copy(backupDir);
```

Особливості методу:

- **Глибоке копіювання (Deep Copy):** Метод не просто створює нову порожню папку, він рекурсивно копіює всі вкладені файли та піддиректорії.
- **Безпека цільового шляху:** Об'єкт `backupDir` не обов'язково має фізично існувати до виклику методу — система створить його автоматично під час копіювання.
- **Уникнення конфліктів:** Якщо цільова папка вже існує і містить файли, операція може викинути виняток, тому важливо перевіряти стан через `backupDir.exists` перед копіюванням.

Переміщення та Перейменування

Для оптимізації дискового простору замість копіювання доцільно використовувати переміщення файлових структур.

1. Переміщення: `move(destination)` Метод переносить директорію у нове місце. *Важлива архітектурна деталь:* Після успішного виконання методу `move`, об'єкт `sourceDir` автоматично оновлює свою властивість `uri`, щоб вказувати на нове місце розташування.

```
const oldDir = new Directory(Paths.cache, 'temp_downloads');
const newLocation = new Directory(Paths.document, 'saved_downloads');

oldDir.move(newLocation);
// Тепер oldDir.uri співпадає з newLocation.uri
```

Переміщення та Перейменування

2. Перейменування: `rename(newName)` Якщо папку не потрібно переміщувати в іншу директорію, а лише змінити її назву (наприклад, `folder1` на `folder2`), використовується метод `.rename()`. Це значно швидша операція на рівні файлової системи, оскільки не потребує зміни фізичних індексів файлів.

Видалення директорій

Управління пам'яттю на рівні файлової системи є зоною відповідальності розробника мобільного додатку.

Метод: `delete()` Виклик цього методу на об'єкті `Directory` ініціює незворотний процес видалення.

Критичні правила роботи з `delete()`:

- **Рекурсивність:** На відміну від стандартних команд ОС (на зразок `rmdir`), які вимагають, щоб папка була порожньою, метод `delete()` в Expo автоматично та безповоротно знищує папку разом із **усіма** вкладеними файлами та підпапками.
- **Очищення кешу:** Найчастіший сценарій використання — періодичне очищення тимчасових даних. Замість ручного перебору файлів достатньо викликати:

```
new Directory(Paths.cache).delete();
```

Примітка: Сама системна директорія кешу відновиться операційною системою при наступному запиті.

Практичний кейс - Рекурсивний сканер директорій

Задача: Написати функцію, яка рахує загальну кількість файлів у заданій папці та всіх її підпапках.

Реалізація на сучасному API:

```
// рекурсивна функція підрахунку файлів
Show usages   new *
const countFilesInDirectory = async (dir) => {
  let fileCount = 0;

  const items = await dir.list();

  for (const item of items) {
    if (item instanceof File) {
      fileCount++;
    }

    if (item instanceof Directory) {
      fileCount += await countFilesInDirectory(item);
    }
  }

  return fileCount;
};
```

Фабричні методи директорії

Об'єкт `Directory` має вбудовані методи для швидкої генерації вкладених об'єктів без необхідності ручного імпорту та склеювання шляхів.

1. Швидке створення підпапки: `createDirectory(name)` Метод автоматично повертає новий екземпляр класу `Directory`, який вказує на вкладену папку.

```
const documentsDir = new Directory(Paths.document);

// Замість: const imagesDir = new Directory(Paths.document, 'images');
const imagesDir = documentsDir.createDirectory('images');
imagesDir.create(); // фізично створюємо на диску
```

Фабричні методи директорії

2. Швидке створення файлу: `createFile(name, mimeType)` Метод генерує екземпляр класу `File` безпосередньо всередині цієї директорії.

```
// Швидко створюємо посилання на новий текстовий файл
const myNote = documentsDir.createFile('note.txt', 'text/plain');

// Тепер можемо відразу в нього писати
myNote.write('Привіт, світе!');
```

Клас `File`: Ініціалізація та базові концепції

Так само як і клас `Directory`, екземпляр класу `File` є віртуальним посиланням на конкретний шлях у файловій системі.

Синтаксис створення об'єкта: Конструктор приймає масив сегментів шляху. Першим аргументом зазвичай виступає об'єкт базової директорії (наприклад, `Paths.document`), а останнім — ім'я файлу з розширенням.

```
import { File, Paths } from 'expo-file-system';

// 1. Посилання на текстовий документ
const noteFile = new File(Paths.document, 'notes', 'meeting.txt');

// 2. Використання фабричного методу з попереднього розділу
const cacheDir = new Directory(Paths.cache);
const tempImage = cacheDir.createFile('avatar.png', 'image/png');
```

Виклик `new File()` створює лише об'єкт у пам'яті JavaScript. Він не гарантує, що файл фізично існує на диску в цей момент.

Властивості (Properties) класу File

Після створення об'єкта розробник отримує доступ до синхронних властивостей, які описують стан файлу.

Ключові властивості:

- **`exists` (boolean):** Показує, чи присутній файл на диску. Також повертає `false`, якщо додаток не має прав на читання цього файлу.
- **`size` (number):** Розмір файлу в байтах. Якщо файлу не існує, повертає `0`.
- **`type` (string):** MIME-тип файлу (наприклад, `application/pdf`, `text/plain`). Якщо система не може визначити тип, повертає порожній рядок.
- **`extension` (string):** Розширення файлу включно з крапкою (наприклад, `.jpg`).
- **`contentUri` (string):** Спеціальний URI (схема `content://`), який генерується системою для безпечної передачі файлу в інші додатки (наприклад, для відправки в месенджери).

```
LOG  URI: file:///data/user/0/host.exp.exponent/files/docs/report.txt
LOG  Exists: true
LOG  Size: 33
LOG  Extension: .txt
LOG  Type: text/plain
LOG  Content URI: content://host.exp.exponent.FileSystemFileProvider/expo_files/docs/report.txt
```

Фізичне створення файлу (create)

Якщо вам потрібно гарантовано створити порожній файл на диску перед початком роботи з ним (наприклад, для подальшого дозаписування байтів або передачі посилання в інший модуль), використовується метод `.create()`.

```
const logFile = new File(Paths.document, 'app.log');

if (!logFile.exists) {
  logFile.create({
    intermediates: true, // Створить папку, якщо її немає
    overwrite: false     // Якщо файл вже є, не буде його затирати
  });
}
```

`overwrite: true | false`: Визначає поведінку у випадку конфлікту. Якщо передати `true`, існуючий файл буде видалено, а на його місці створено новий порожній. За замовчуванням — `false`.

Запис текстових даних (write)

У сучасному SDK метод `writeAsStringAsync` було видалено. Замість нього використовується універсальний поліморфний метод `.write()`, який вміє працювати як з текстом, так і з бінарними даними.

Синтаксис запису:

```
const configFile = new File(Paths.document, 'settings.json');

// Метод write є атомарним і повністю перезаписує вміст файлу
configFile.write('{"theme": "dark", "notifications": true}');
```

Особливості:

1. **Атомарність:** Операція запису відбувається цілком. Якщо під час запису станеться збій живлення або переповнення пам'яті, файл або запишеться повністю, або не запишеться взагалі (запобігає пошкодженню даних).
2. **Автоматичне створення:** На відміну від старих API, якщо ви викликаєте `.write()` на файлі, якого ще не існує, система **автоматично** створить його. Попередньо викликати `.create()` не потрібно.

Читання тексту та обробка JSON

Зчитування даних з локального диска — одна з найчастіших операцій.

Методи читання:

- **`text()` (Асинхронний):** Повертає `Promise<string>`. Не блокує інтерфейс користувача (UI-потік). Є стандартом для мобільної розробки.
- **`textSync()` (Синхронний):** Блокує потік до завершення читання. Використовувати дозволяється *лише* для дуже маленьких файлів (до кількох кілобайт), наприклад, при стартовій ініціалізації додатка.

Зчитування конфігурації (JSON)

```
async function loadUserSettings() {
  const settingsFile = new File(Paths.document, 'settings.json');

  if (!settingsFile.exists) {
    return { theme: 'light' }; // Повертаємо дефолтні значення
  }

  try {
    const jsonString = await settingsFile.text();
    return JSON.parse(jsonString); // Перетворення рядка в JS-об'єкт
  } catch (error) {
    console.error("Помилка парсингу або читання файлу:", error);
    return null;
  }
}
```

Інтроспекція файлу та перевірка цілісності

Перед відправкою файлу на сервер або після його завантаження необхідно перевірити його технічні характеристики та цілісність. Для цього використовується метод `.info()`.

Метод: `info(options?)` Повертає об'єкт `FileInfo`, який містить розширені дані про файл.

```
// записуємо тестові дані
if (!documentFile.exists) {
  await documentFile.write("Test contract content");
}

// отримуємо технічну інформацію
const metadata = await documentFile.info({ md5: true });

console.log("File URI:", documentFile.uri);
console.log("Size:", metadata.size, "bytes");
console.log("Last modified:", new Date(metadata.modificationTime));
console.log("MD5 hash:", metadata.md5);
```

MD5: Параметр `{ md5: true }` змушує систему вирахувати криптографічний хеш файлу. Це критично важливо для перевірки цілісності при завантаженні файлів через нестабільну мобільну мережу. Якщо MD5 локального файлу співпадає з MD5 на сервері — файл завантажився без "битих" байтів.

Релокація даних: Копіювання та Переміщення

Об'єкти класу `File` мають вбудовані методи для зміни свого фізичного розташування на накопичувачі.

1. Копіювання: `copy(destination)` Створює точну копію файлу. Цільовим об'єктом (`destination`) може бути як нова директорія, так і конкретний новий файл (якщо потрібно змінити ім'я під час копіювання).

```
const originalPhoto = new File(Paths.cache, 'photo_123.jpg');
const backupDir = new Directory(Paths.document, 'backups');

// Копіюємо файл у папку backups (ім'я збережеться)
originalPhoto.copy(backupDir);
```

Релокація даних: Копіювання та Переміщення

2. Переміщення: `move(destination)` Переносить файл. Як і у випадку з директоріями, після успішного переміщення, властивість `uri` оригінального об'єкта `File` автоматично оновлюється. Це найшвидший спосіб перенести тимчасовий файл з `Paths.cache` у постійне сховище `Paths.document`.

Безпечне видалення файлів

Управління життєвим циклом файлу завершується його видаленням. На відміну від директорій, видалення одного файлу є менш ресурсоємною, але не менш відповідальною операцією.

Метод: `delete()` Синхронно або асинхронно ініціює видалення файлу з фізичного диска.

```
const tempFile = new File(Paths.cache, 'temp_session.json');

try {
  if (tempFile.exists) {
    tempFile.delete();
    console.log("Файл успішно видалено");
  }
} catch (error) {
  console.error("Помилка видалення (можливо файл заблоковано ОС):", error.messag
}
```

Клас `Paths`: Властивості системних директорій

Об'єкт `Paths` — це глобальний сервіс SDK, який надає стандартизований доступ до системних директорій мобільного застосунку. Він інкапсулює функціональність утиліт роботи з файловими шляхами та повертає готові екземпляри класу `Directory`, що відповідають ключовим каталогам середовища виконання.

Основне призначення `Paths` — забезпечити **безпечний та уніфікований доступ до службових папок додатка** (наприклад, `document`, `cache`, `temporary`) без необхідності вручну формувати файлові шляхи або працювати з платформозалежними URI.

Основні властивості директорій:

- **`document`**
  - **Тип:** `Directory`
  - **Опис:** Місце для зберігання постійних файлів користувача. Дані в цій директорії надійно захищені від автоматичного видалення операційною системою.
- **`cache`**
  - **Тип:** `Directory`
  - **Опис:** Тимчасове сховище. Система має право самостійно видалити вміст цієї папки, коли на пристрої виникає дефіцит вільного місця.
- **`bundle`**
  - **Тип:** `Directory`
  - **Опис:** Каталог додатка. Зберігає статичні ресурси (assets), що постачаються разом із скомпільованою програмою (Read-Only).
- **`appleSharedContainers`** (Тільки для iOS)
  - **Тип:** `Record<string, Directory>`
  - **Опис:** Словник спільних контейнерів (App Groups) для обміну даними між основним додатком та його розширеннями (наприклад, віджетами).

Персистентні дані: **Paths.document**

Це головна "база" для зберігання інформації, яка критично важлива для користувача і не повинна зникнути після перезавантаження телефону.

- **Властивість:** `Paths.document`
- **Тип:** Екземпляр класу `Directory`
- **Ключова особливість (Захист):** Дані в цій директорії надійно захищені від автоматичного видалення операційною системою. Якщо файл потрапив сюди, він зникне лише у двох випадках: якщо ви самі викличете `file.delete()`, або якщо користувач повністю видалить ваш додаток з пристрою.
- **Резервне копіювання:** За замовчуванням (на iOS та Android), вміст цієї папки автоматично потрапляє у хмарні бекапи користувача (iCloud / Google Drive).

Що тут зберігати: Файли локальних баз даних (SQLite), збережені користувачем PDF-документи, завантажені офлайн-карти, критичні налаштування додатку.

```
import { Paths, Directory } from 'expo-file-system';

// Створюємо підпапку для звітів користувача у захищеному сховищі
const reportsDir = new Directory(Paths.document, 'monthly_reports');

if (!reportsDir.exists) {
  reportsDir.create();
}

// Тепер можемо безпечно створювати файли всередині reportsDir
const newReport = reportsDir.createFile('january.pdf');
```

Волатильні (тимчасові) дані: **Paths.cache**

Сучасні додатки постійно завантажують з мережі аватарки, обкладинки відео, тимчасові JSON-відповіді. Щоб додаток не збільшився до гігантських розмірів, використовується директорія кешу.

- **Властивість:** `Paths.cache`
- **Тип:** Екземпляр класу `Directory`
- **Ключова особливість:** Операційна система має повне право **самостійно, без попередження** видалити вміст цієї папки, коли на пристрої виникає дефіцит вільного місця. Також ці дані ніколи не бекапляться в хмару.

Архітектурне правило (Best Practice): Ніколи не зберігайте тут унікальні дані. Будь-який запит до файлу в `Paths.cache` має супроводжуватися перевіркою `if (file.exists)`. Якщо система видалила файл — ваш код має завантажити його повторно.

```
import { Paths, File } from 'expo-file-system';

async function getCachedAvatar(userId, url) {
  const avatarFile = new File(Paths.cache, `avatar_${userId}.jpg`);

  // Обов'язкова перевірка: чи не видалила ОС цей файл?
  if (avatarFile.exists) {
    return avatarFile.uri; // Повертаємо локальний шлях
  }

  // Якщо файлу немає (або його видалили) - завантажуємо знову
  const response = await fetch(url);
  const buffer = await response.arrayBuffer();
  avatarFile.write(new Uint8Array(buffer));

  return avatarFile.uri;
}
```

Ресурси етапу компіляції: **Paths.bundle**

Іноді додаток повинен постачатися з уже готовими файлами, запакованими прямо в інсталятор (`.apk` для Android або `.ipa` для iOS). Це можуть бути початкові бази даних міст, складні 3D-моделі для ігор або локалізовані JSON-файли.

- **Властивість:** `Paths.bundle`
- **Тип:** Екземпляр класу `Directory`
- **Критичне обмеження (Read-Only):** Ця директорія є **строго доступною лише для читання**.

Архітектурна пастка: Спроба викликати метод `.write()` або `.create()` для файлу всередині `Paths.bundle` гарантовано призведе до Crash-помилки. Якщо вам потрібно модифікувати стартову базу даних (наприклад, додати нові записи), ви повинні спочатку скопіювати її з `bundle` у `document`, і працювати вже з копією.

```
import { Paths, File } from 'expo-file-system';

async function loadInitialCities() {
  // Звертаємось до файлу, який ми поклали в папку assets перед компіляцією
  const citiesDb = new File(Paths.bundle, 'assets/cities.json');

  if (citiesDb.exists) {
    // Ми можемо його читати!
    const jsonString = await citiesDb.text();
    return JSON.parse(jsonString);
  }
  return [];
}
```

Міжпроцесна взаємодія: **Paths.appleSharedContainers**

Ця властивість є специфічною виключно для екосистеми iOS (на Android вона поверне порожній об'єкт або `undefined`).

- **Властивість:** `Paths.appleSharedContainers`
- **Тип:** `Record<string, Directory>` (Словник об'єктів директорій).
- **Проблема, яку вирішує:** В iOS кожен додаток і його віджет (або розширення для Apple Watch) працюють у різних, суворо ізольованих "пісочницях". Основний додаток не бачить папку `document` віджета, а віджет не бачить папку додатку.
- **Рішення:** Ця властивість дозволяє отримати доступ до спеціальної спільної папки (контейнера), куди обидва процеси можуть писати та читати дані одночасно.

```
import { Paths, File } from 'expo-file-system';

// 'group.com.yourcompany.app' - це ідентифікатор, який ви налаштовуєте в Apple [
const sharedDir = Paths.appleSharedContainers['group.com.yourcompany.app'];

if (sharedDir) {
  // Цей файл зможе прочитати як ваш основний додаток, так і ваш iOS-віджет
  const widgetData = new File(sharedDir, 'widget_state.json');
  widgetData.write('{"battery": 80, "status": "active"}');
} else {
  console.log("App Groups не налаштовані або це не iOS пристрій");
}
```

Апаратні властивості: Аналіз накопичувача

Для запобігання фатальним помилкам (Crash) під час запису великих обсягів даних, клас `Paths` надає синхронний доступ до інформації про фізичний стан внутрішньої пам'яті смартфона.

Властивості дискового простору:

- **`totalDiskSpace`**
  - **Тип:** `number`
  - **Опис:** Представляє загальний фізичний обсяг внутрішньої пам'яті пристрою. Вимірюється у байтах.
- **`availableDiskSpace`**
  - **Тип:** `number`
  - **Опис:** Представляє фактичний доступний (вільний) простір у внутрішній пам'яті на момент виклику. Також вимірюється у байтах.

Правило безпечного коду: Завжди порівнюйте очікуваний розмір файлу (наприклад, з HTTP-заголовка `Content-Length`) із `availableDiskSpace` перед ініціалізацією завантаження.

Апаратний моніторинг: Властивість **totalDiskSpace**

Перш ніж додаток почне працювати з файловою системою, корисно розуміти фізичні межі пристрою користувача. Для цього клас `Paths` надає синхронний доступ до інформації про загальну ємність накопичувача.

- **Синтаксис:** `Paths.totalDiskSpace`
- **Тип даних:** `number`
- **Опис:** Повертає загальний фізичний обсяг внутрішньої пам'яті пристрою. Значення завжди представлено у **байтах** і включає простір, зайнятий операційною системою (iOS/Android) та іншими додатками.

Практичне застосування (Відображення в UI): Користувачам незручно читати довгі числа в байтах. Найчастіший сценарій використання цієї властивості — конвертація байтів у Гігабайти (ГБ) для відображення на екрані "Налаштувань" або "Профілю", щоб показати, яку частку пам'яті займає ваш додаток відносно всього диска.

```
useEffect(() => {

  Show usages  new *
  function getDeviceCapacityInGB() {

    // отримуємо загальний обсяг диску у байтах
    const bytes = Paths.totalDiskSpace;

    // конвертуємо у гігабайти
    const gigabytes = bytes / (1024 ** 3);

    return gigabytes.toFixed(2);
  }

  const total = getDeviceCapacityInGB();

  console.log(`Total device storage: ${total} GB`);

}, []);
```

Запобігання **Crash**-помилкам: **availableDiskSpace**

Найпоширеніша причина фатального збою під час роботи з файлами — помилка ENOSPC (Error No Space Left On Device). Вона виникає, коли додаток намагається записати файл, розмір якого перевищує залишок вільного місця на диску.

- **Синтаксис:** `Paths.availableDiskSpace`
- **Тип даних:** `number`
- **Опис:** Повертає фактичний обсяг вільного простору (у байтах), який доступний для запису в дану мілісекунду.

Правило безпечного програмування: Значення `availableDiskSpace` є динамічним. Система або фонові процеси можуть зайняти місце будь-якої миті. Тому перевірка вільного місця має відбуватися **безпосередньо перед** стартом завантаження великого файлу або записом бази даних.

```
useEffect(() => {

  // отримуємо доступний простір у байтах
  const freeBytes = Paths.availableDiskSpace;

  // переводимо у гігабайти
  const freeGB = (freeBytes / (1024 ** 3)).toFixed(2);

  console.log("Available disk space:", freeBytes, "bytes");
  console.log("Available disk space:", freeGB, "GB");

}, []);

return (
    <View style={{flex:1, justifyContent:"center", alignItems:"center"}}>
      <Text>Check free disk space</Text>
    </View>
);
```

Утиліти деконструкції шляхів

Крім системних папок, `Paths` містить набір синхронних методів для аналізу рядкових шляхів. **Більшість методів як параметр `path` приймають не лише рядок, а й готові об'єкти `File` або `Directory`.**

Утиліти деконструкції шляхів

`basename(path, ext?)`

- **Параметри:** `path` (`string | File | Directory`), `ext` (`string`, необов'язково).
- **Повертає:** `string` (базову назву файлу або папки). Якщо передати `ext` (наприклад, `".txt"`), воно буде відокремлене від результату.

`dirname(path)`

- **Параметри:** `path` (`string | File | Directory`).
- **Повертає:** `string` (назву батьківського каталогу за вказаним шляхом).

`extname(path)`

- **Параметри:** `path` (`string | File | Directory`).
- **Повертає:** `string` (розширення файлу, наприклад, `".pdf"`).

Утиліти парсингу та метаданих шляху

Для складнішого аналізу структури URI використовуються методи парсингу.

`parse(path)`

- **Параметри:** `path` (`string | File | Directory`).
- **Повертає:** Об'єкт. Розбиває шлях на його складові компоненти:

```
{
  root: string, // Кореневий префікс (напр., 'file://')
  dir: string,  // Шлях до директорії
  base: string, // Повне ім'я файлу з розширенням
  ext: string,  // Розширення
  name: string  // Ім'я файлу без розширення
}
```

`info(...uris)`

- **Параметри:** `...uris` (`string[]`).
- **Повертає:** `PathInfo`. Об'єкт, який вказує метадані шляху (зокрема, чи є вказаний URI каталогом чи файлом).

Метод **Paths.basename()**

Часто в додатку ми маємо повний абсолютний шлях до файлу, але для відображення в інтерфейсі (наприклад, у списку завантажень) нам потрібна лише сама назва файлу. Для цього використовується метод витягу базового імені.

- **Синтаксис:** `Paths.basename(path, ext?)`
- **Параметри:**
  - `path` (`string | File | Directory`): Шлях, який потрібно проаналізувати.
  - `ext` (`string`, необов'язково): Конкретне розширення, яке ви хочете автоматично "відрізати" від результату.
- **Повертає:** `string` (кінцевий сегмент шляху).

```javascript
import { Paths } from 'expo-file-system';

const fullUri = 'file:///data/user/documents/reports/annual_report.pdf';

// 1. Отримання повного імені файлу
const fileName = Paths.basename(fullUri);
console.log(fileName); // Вивід: "annual_report.pdf"

// 2. Отримання імені без розширення (передаємо розширення другим параметром)
const cleanName = Paths.basename(fullUri, '.pdf');
console.log(cleanName); // Вивід: "annual_report"
```

Метод **Paths.dirname()**

Якщо ви хочете створити файл за певним шляхом, хорошою практикою є спочатку перевірити, чи існує папка, в якій він має знаходитись. Метод `dirname` дозволяє легко відсікти ім'я файлу і отримати шлях до його батьківської директорії.

- **Синтаксис:** `Paths.dirname(path)`
- **Параметри:** `path` (`string | File | Directory`).
- **Повертає:** `string` (шлях до директорії, що містить кінцевий файл або папку).

Приклад у коді:

```
import { Paths, Directory } from 'expo-file-system';

const targetFilePath = 'file:///app/downloads/media/video.mp4';

// Отримуємо шлях до батьківської папки
const parentDirPath = Paths.dirname(targetFilePath);
console.log(parentDirPath); // Вивід: "file:///app/downloads/media"

// Тепер можемо безпечно перевірити чи створити цю папку
const parentDir = new Directory(parentDirPath);
if (!parentDir.exists) {
  parentDir.create();
}
```

Визначення типу: Метод **Paths.extname()**

При завантаженні файлів на сервер (Upload) або використанні системного шерингу (Sharing) вам часто потрібно вказати MIME-тип файлу (наприклад, `image/jpeg` або `application/pdf`). Щоб зрозуміти, з яким файлом ми працюємо, необхідно витягти його розширення.

- **Синтаксис:** `Paths.extname(path)`
- **Параметри:** `path` (`string | File | Directory`).
- **Повертає:** `string` (розширення файлу, включаючи крапку).

Приклад у коді:

```
useEffect(() => {

  const fullUri =
      "file:///data/user/0/com.app/files/reports/annual_report.pdf";

  // отримуємо повне ім'я файлу
  const fileName = Paths.basename(fullUri);

  console.log("File name:", fileName);
}, []);
```

Деконструкція: Метод **Paths.parse()**

Коли архітектура вимагає одночасного знання і імені файлу, і його розширення, і його папки — викликати три попередні методи окремо неефективно. Метод `parse` робить це за один прохід, розбиваючи шлях на структурний об'єкт.

- **Синтаксис:** `Paths.parse(path)`
- **Параметри:** `path` (`string | File | Directory`).
- **Повертає:** Об'єкт з ключами `root`, `dir`, `base`, `ext`, `name`.

```javascript
useEffect(() => {

  const myPath = "file:///var/mobile/Containers/Data/report.xlsx";

  const parsed = Paths.parse(myPath);

  console.log("Full object:", parsed);

  console.log("Root:", parsed.root);
  console.log("Directory:", parsed.dir);
  console.log("File name:", parsed.base);
  console.log("Name without extension:", parsed.name);
  console.log("Extension:", parsed.ext);

}, []);
```

Інтроспекція метаданих: Метод **Paths.info()**

Іноді вам потрібно прийняти рішення на основі масиву шляхів (наприклад, відфільтрувати лише директорії з наданого списку URI), але ви не хочете витрачати ресурси пам'яті на створення екземплярів `new File()` або `new Directory()` для кожного з них.

- **Синтаксис:** `Paths.info(...uris)`
- **Параметри:** `...uris` (`string[]` — масив рядкових шляхів).
- **Повертає:** Об'єкт типу `PathInfo` (містить метадані, що дозволяють відрізнити файл від директорії на рівні системи).

```
useEffect(() => {

  const urisToCheck = [
    "file:///data/documents/folder_A",
    "file:///data/documents/note.txt"
  ];

  // отримуємо інформацію про кілька шляхів
  const infoResult = Paths.info(...urisToCheck);

  console.log("Info result:", infoResult);

}, []);
```

Утиліти побудови та маршрутизації

Для безпечного створення нових шляхів без ризику помилитися у слешах (`/`) використовуються методи маршрутизації.

- **`join(...paths)`**
  - **Параметри:** `...paths` масив `(string | File | Directory)[]`.
  - **Повертає:** `string`. Об'єднує всі передані сегменти в єдиний валідний шлях.
- **`normalize(path)`**
  - **Параметри:** `path` `(string | File | Directory)`.
  - **Повертає:** `string`. Нормалізує шлях (вирішує проблеми з `..` та зайвими `/`).
- **`isAbsolute(path)`**
  - **Повертає:** `boolean`. Перевіряє, чи є шлях абсолютним (`true`) чи відносним (`false`).
- **`relative(from, to)`**
  - **Параметри:** `from` (базовий шлях), `to` (відносний шлях). Обидва типу `string | File | Directory`.
  - **Повертає:** `string`. Перетворює відносний шлях на абсолютний, відштовхуючись від базового.

Утиліти маршрутизації: **Paths.join() (**Безпечне об'єднання**)**

Найголовніше правило роботи з будь-якою файловою системою: **ніколи не створюйте шляхи власноруч через конкатенацію рядків** (наприклад, `folderPath + '/' + fileName`). Це майже гарантовано призводить до помилок (подвійних слешів `//` або їх нестачі), які спричиняють збій програми.

- **Синтаксис:** `Paths.join(...paths)`
- **Параметри:** Необмежена кількість сегментів `(string | File | Directory)[]`.
- **Повертає:** `string` (єдиний валідний шлях).

Як це працює під капотом: Метод бере всі передані сегменти, автоматично видаляє зайві розділювачі на їхніх стиках і гарантує, що шлях буде синтаксично правильним для поточної операційної системи.

```
useEffect(() => {

  const baseUri = "file:///data/user/0/com.app/documents/";
  const folder = "/images/";
  const file = "avatar.png";

  // невірний спосіб (можуть бути подвійні слеші)
  const badPath = baseUri + folder + file;

  // правильний спосіб
  const safePath = Paths.join(baseUri, folder, file);

  console.log("Bad path:", badPath);
  console.log("Safe path:", safePath);

}, []);
```

Утиліти маршрутизації: **Paths.normalize() (**Очищення шляху**)**

У процесі роботи додаток може отримувати шляхи від сторонніх API, розпаковувати ZIP-архіви або приймати користувацький ввід. Такі шляхи часто містять інструкції навігації: `.` (поточна директорія) або `..` (піднятися на рівень вище).

- **Синтаксис:** `Paths.normalize(path)`
- **Параметри:** `path` (`string | File | Directory`).
- **Повертає:** `string` (очищений та спрощений шлях).

Призначення та безпека: Цей метод вирішує інструкції навігації та прибирає дублюючі слеші. Крім того, він є важливим інструментом безпеки для запобігання атакам типу **Directory Traversal** (коли зловмисник намагається вийти за межі дозволеної папки за допомогою `../../`).

```
useEffect(() => {

  // "брудний" шлях з зайвими переходами та слешами
  const dirtyPath =
      "file:///data/users/admin/../guest/./logs//crash.txt";

  console.log("Dirty path:", dirtyPath);

  // нормалізація
  const cleanPath = Paths.normalize(dirtyPath);

  console.log("Clean path:", cleanPath);

}, []);
```

```
LOG  Dirty path: file:///data/users/admin/../guest/./logs//crash.txt
LOG  Clean path: file:///data/users/guest/logs/crash.txt
```

Утиліти маршрутизації: **Paths.isAbsolute() (**Контекст шляху**)**

Для того щоб операційна система смартфона могла фізично прочитати або записати файл, їй потрібен **абсолютний шлях** — тобто шлях, який починається від самого кореня файлової системи (наприклад, з `file:///` або `/`). Відносні шляхи (наприклад, `images/photo.jpg`) ОС не зрозуміє, оскільки не знає, від якої точки вести відлік.

- **Синтаксис:** `Paths.isAbsolute(path)`
- **Параметри:** `path` (`string | File | Directory`).
- **Повертає:** `boolean` (`true` — абсолютний, `false` — відносний).

Архітектурне використання: Метод використовується як "запобіжник" у функціях-обгортках. Перед передачею шляху в метод `File.write()`, ви можете перевірити, чи є він абсолютним, і якщо ні — програмно додати до нього базову директорію (`Paths.document`).

```javascript
useEffect(() => {

  const path1 = "/var/mobile/Containers/Data/file.txt";
  const path2 = "file:///data/user/0/com.app/files/photo.jpg";
  const path3 = "assets/icons/logo.png";

  console.log("Path1 is absolute:", Paths.isAbsolute(path1));
  console.log("Path2 is absolute:", Paths.isAbsolute(path2));
  console.log("Path3 is absolute:", Paths.isAbsolute(path3));

}, []);
```

Утиліти маршрутизації: **Paths.relative() (**Відносні маршрути**)**

На відміну від `isAbsolute`, цей метод обчислює **різницю** між двома абсолютними шляхами. Він відповідає на запитання: *"Які інструкції навігації (`..`, `/`) потрібно застосувати, щоб з папки А потрапити у папку Б?"*

- **Синтаксис:** `Paths.relative(from, to)`
- **Параметри:** `from` (базовий/стартовий шлях), `to` (цільовий шлях).
- **Повертає:** `string` (відносний шлях-інструкцію).

Критичний сценарій використання (Переносимість даних): Абсолютні шляхи (URI) ваших папок `document` змінюються щоразу, коли користувач оновлює додаток з App Store / Google Play (ОС генерує новий UUID-хеш для "пісочниці"). Якщо ви збережете абсолютний шлях у локальну базу даних (SQLite), після оновлення додатку він перестане працювати. Тому в базі зберігають *відносні шляхи*, а генерують їх за допомогою `relative()`.

```javascript
useEffect(() => {

  const documentDir = "file:///data/app_hash_123/documents/";
  const imageFile   = "file:///data/app_hash_123/documents/photos/cat.jpg";

  // обчислюємо відносний шлях
  const relativePath = Paths.relative(documentDir, imageFile);

  console.log("Relative path:", relativePath);

  // відновлення повного шляху
  const fullPath = Paths.join(documentDir, relativePath);

  console.log("Restored full path:", fullPath);

}, []);
```

Робота з мультимедіа: Base64 Кодування

Коли ми працюємо з зображеннями або аудіо, метод `text()` не підходить, оскільки бінарні дані неможливо коректно перетворити у звичайний UTF-8 рядок. Першим способом роботи з бінарними даними в React Native є кодування Base64.

Що таке Base64? Це стандарт перетворення бінарних байтів у безпечний текстовий рядок (ASCII), який можна передавати через JSON або вбудовувати безпосередньо в UI.

Методи: `base64()` та `base64Sync()`

```
const avatarFile = new File(Paths.document, 'avatar.png');

// Отримуємо рядок Base64
const base64String = await avatarFile.base64();

// Використання Data URI в компоненті React Native <Image>
const imageSource = { uri: `data:image/png;base64,${base64String}` };
```

Недолік методу: Рядок Base64 займає на 33% більше пам'яті (RAM), ніж оригінальний файл. Не рекомендується для файлів розміром більше 10-15 МБ.

Низькорівнева робота з байтами (Uint8Array)

Для сучасних криптографічних задач, роботи з потоковим відео або специфічними бінарними протоколами (наприклад, Bluetooth LE або IoT-пристроями), необхідно мати прямий доступ до байтів пам'яті.

Новий стандарт Web API в Expo: Сучасний клас `File` імплементує інтерфейс `Blob`, що дозволяє працювати з даними у форматі типізованих масивів (Typed Arrays) JavaScript.

Методи: `bytes()` та `bytesSync()` Повертають об'єкт `Uint8Array` (масив 8-бітних цілих чисел без знака).

Низькорівнева робота з байтами **(Uint8Array)**

```
const encryptedFile = new File(Paths.document, 'secret.dat');

// Завантажуємо файл у пам'ять як масив байтів
const byteData = await encryptedFile.bytes();

// byteData[0] – це перший байт файлу (від 0 до 255)
console.log(`Перший байт файлу: ${byteData[0]}`);

// Запис масиву байтів назад у файл
const newFile = new File(Paths.document, 'decrypted.dat');
newFile.write(byteData);
```

Перевага: Це найефективніший спосіб маніпуляції даними у JavaScript, що не створює надлишкового навантаження на оперативну пам'ять (на відміну від Base64).

Проблема великих файлів та переповнення пам'яті (OOM)

Методи `.text()`, `.base64()` та `.bytes()`, які ми розглянули раніше, мають суттєвий архітектурний недолік: вони завантажують **весь** вміст файлу в оперативну пам'ять (RAM) смартфона одночасно.

Що таке Out Of Memory (OOM) Crash? Операційні системи (особливо iOS) суворо лімітують обсяг оперативної пам'яті для кожного додатка. Якщо ваш додаток спробує прочитати відеофайл розміром 1 ГБ через `await file.bytes()`, рушій JavaScript перевищить ліміт пам'яті, і ОС примусово "вб'є" процес додатка без попередження.

Рішення: Для роботи з великими файлами (відео, великі архіви, об'ємні бази даних) сучасний API пропонує два підходи:

1. Низькорівневе керування через клас **`FileHandle`**.
2. Використання стандартних Web **Потоків (Streams)**.

Клас FileHandle: Низькорівневий доступ

Клас `FileHandle` надає можливість читати та записувати дані у файл поступово, невеликими порціями (чанками), контролюючи позицію курсора (offset).

Відкриття файлу: Щоб отримати об'єкт `FileHandle`, необхідно викликати метод `.open()` на екземплярі `File`.

```
const hugeVideo = new File(Paths.document, 'movie.mp4');

// Відкриваємо файл для низькорівневого доступу
const handle = hugeVideo.open();

console.log(`Розмір відкритого файлу: ${handle.size} байт`);
console.log(`Поточна позиція курсора: ${handle.offset}`); // За замовчуванням 0
```

Клас **FileHandle**: Низькорівневий доступ

Критичне правило (Управління ресурсами): Кожен відкритий `FileHandle` блокує файл на рівні операційної системи (інші процеси не можуть його видалити). Після завершення роботи ви **зобов'язані** викликати метод **`handle.close()`**, щоб звільнити пам'ять та зняти блокування. Зазвичай це робиться у блоці `finally`.

Читання та запис чанками

За допомогою `FileHandle` ми можемо **читати файл частинами(наприклад, по 1 мегабайту),** обробляти їх і звільняти пам'ять перед читанням наступної частини.

1. Читання (`readBytes`):

```
const handle = hugeVideo.open();
try {
  // Читаємо перші 1024 байти (1 КБ)
  const firstChunk = handle.readBytes(1024);
  // Властивість offset автоматично збільшилась на 1024
} finally {
  handle.close();
}
```

Читання та запис чанками

2. Запис (`writeBytes`): Дозволяє записувати `Uint8Array` масиви починаючи з поточного `offset`. Якщо `offset` менший за розмір файлу — дані будуть перезаписані поверх існуючих. Якщо `offset` дорівнює розміру файлу — дані будуть додані в кінець.

Потоки та сумісність з **Web API**

Клас `File` в сучасному Expo SDK імплементує глобальний Web-інтерфейс `Blob`. Це означає, що ви можете використовувати ті ж самі механізми потоків (Streams), що й у браузерному JavaScript або Node.js.

Метод: `stream()` або `readableStream()` Повертає стандартний `ReadableStream`, який **дозволяє обробляти дані в міру їх надходження**.

Замість того щоб **читати весь файл одразу в пам'ять**, ми читаємо його **по частинах (chunks)** через **Stream**.

```
LOG  Прочитано чанк: 1024 байт
LOG  Прочитано чанк: 1024 байт
LOG  Прочитано чанк: 1024 байт
LOG  Прочитано чанк: 1024 байт
LOG  Прочитано чанк: 1024 байт
LOG  Прочитано чанк: 1024 байт
LOG  Прочитано чанк: 1024 байт
LOG  Прочитано чанк: 1024 байт
LOG  Прочитано чанк: 1024 байт
LOG  Прочитано чанк: 1024 байт
LOG  Прочитано чанк: 1024 байт
LOG  Прочитано чанк: 280 байт
LOG  Читання завершено
```

```
const bigLog = new File(Paths.document, 'system_events.log');

// Отримуємо потік і його "читача" (reader)
const stream = bigLog.stream();
const reader = stream.getReader();

while (true) {
  const { done, value } = await reader.read();

  if (done) {
    console.log("Читання файлу завершено!");
    break;
  }

  // value - це Uint8Array чанк даних
  console.log(`Прочитано чанк розміром: ${value.byteLength} байт`);
}
```

Перевага: Цей підхід добре поєднується з сучасними бібліотеками для завантаження на сервер (`fetch`), які приймають `ReadableStream` як тіло запиту (`body`).

Ефективне дозаписування логів

У legacy API `FileSystem.writeAsStringAsync` не було надійної можливості додати рядок у кінець існуючого файлу (треба було спочатку зчитати весь файл у пам'ять, додати рядок, і перезаписати все назад). З появою `FileHandle` це робиться миттєво і без витрат RAM.

```
async function appendLog(message) {
  const logFile = new File(Paths.document, 'app_crash.log');

  // Якщо файлу ще немає - створюємо
  if (!logFile.exists) logFile.create();

  const handle = logFile.open();

  try {
    // 1. Переміщуємо курсор у самий кінець файлу
    handle.offset = handle.size;

    // 2. Конвертуємо наш текст у масив байтів (Uint8Array)
    const encoder = new TextEncoder();
    const bytesToWrite = encoder.encode(`[${new Date().toISOString()}] ${message}

    // 3. Записуємо байти
    handle.writeBytes(bytesToWrite);

  } catch(error) {
    console.error("Помилка запису логу:", error);
  } finally {
    // 4. Обов'язково закриваємо файл
    handle.close();
  }
}
```

Завантаження файлів з мережі

У застарілому API для завантаження файлів використовувався специфічний метод `FileSystem.downloadAsync()`. У сучасному SDK архітектура змінилася: оскільки клас `File` імплементує Web-стандарт `Blob`, Expo рекомендує використовувати стандартний `fetch` (або оптимізований пакет `@expo/fetch`).

Сучасний патерн завантаження: Ми робимо мережевий запит, отримуємо бінарні дані (ArrayBuffer) і записуємо їх напряму в локальний файл.

```
import { File, Paths } from 'expo-file-system';

async function downloadReport(url) {
  try {
    // 1. Завантажуємо дані в пам'ять
    const response = await fetch(url);
    const buffer = await response.arrayBuffer();

    // 2. Створюємо посилання на локальний файл
    const localFile = new File(Paths.document, 'report.pdf');

    // 3. Записуємо байти (Uint8Array) у файл
    localFile.write(new Uint8Array(buffer));

    console.log(`Файл збережено за адресою: ${localFile.uri}`);
  } catch (error) {
    console.error("Помилка мережі або запису:", error);
  }
}
```

Відправка файлів на сервер (Upload)

Для завантаження локальних файлів на віддалений сервер старий метод `FileSystem.uploadAsync` також визнано `deprecated`.

Рішення: Використання `FormData` та Blob API
Оскільки екземпляр класу `File` тепер є повноцінним `Blob` об'єктом, його можна напряму додавати у `FormData`, що робить код ідентичним до веб-розробки.

```
async function uploadAvatar() {
  const avatarFile = new File(Paths.document, 'avatar.jpg');

  if (!avatarFile.exists) return;

  // Створюємо форму для відправки (multipart/form-data)
  const formData = new FormData();

  // Додаємо об'єкт File напряму!
  formData.append('profile_picture', avatarFile, avatarFile.name);

  // Відправляємо стандартним fetch
  const response = await fetch('https://api.yourdomain.com/upload', {
    method: 'POST',
    body: formData,
    headers: {
      'Authorization': 'Bearer YOUR_TOKEN'
    }
  });

  console.log("Статус відправки:", response.status);
}
```

Інтеграція з системними Пікерами

Коли користувач обирає файл через `expo-document-picker` або `expo-image-picker`, система повертає тимчасовий URI.

Архітектурна проблема: Тимчасові файли (з папок `NSTemporaryDirectory` на iOS або `cache` на Android) можуть бути видалені операційною системою у будь-який момент.

Правило безпеки: Одразу після вибору файлу його необхідно скопіювати у вашу персистентну директорію (`Paths.document`).

```
import * as DocumentPicker from 'expo-document-picker';
import { File, Paths } from 'expo-file-system';

async function pickAndSaveDocument() {
  const result = await DocumentPicker.getDocumentAsync();

  if (!result.canceled) {
    // 1. Отримуємо тимчасовий URI від пікера
    const pickedUri = result.assets[0].uri;

    // 2. Створюємо об'єкт вихідного файлу
    const pickedFile = new File(pickedUri);

    // 3. Створюємо об'єкт призначення в надійній папці
    const savedFile = new File(Paths.document, pickedFile.name);

    // 4. Копіюємо
    pickedFile.copy(savedFile);

    console.log("Документ надійно збережено:", savedFile.uri);
  }
}
```

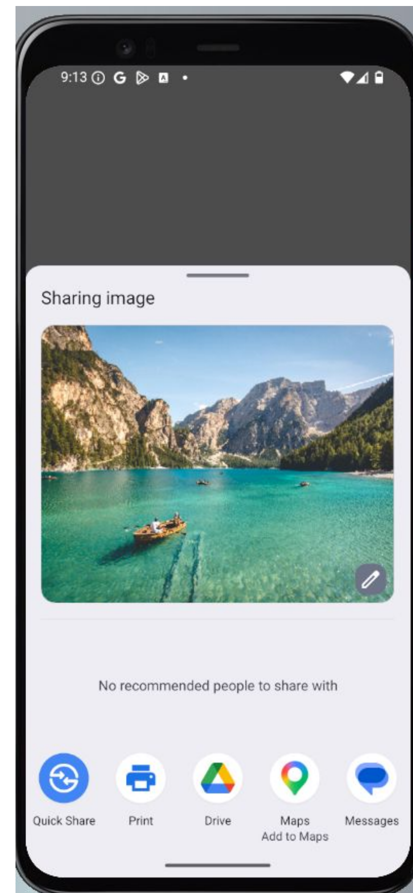
Експорт та Шеринг даних

Якщо ваш додаток згенерував звіт (PDF або Excel), користувач захоче надіслати його в Telegram, на пошту або зберегти на Google Drive.

Взаємодія `expo-file-system` та `expo-sharing`: Для безпечної передачі файлу іншим додаткам не можна використовувати прямий шлях (`file://`), оскільки інші додатки не мають доступу до вашої "пісочниці". Необхідно використовувати властивість **`contentUri`**.

Експорт та Шеринг даних

```
Show usages  new *
async function shareImage() {

  const file = new File(Paths.document, "images/nature.jpg");

  if (file.exists && await Sharing.isAvailableAsync()) {

    await Sharing.shareAsync(file.uri, {
      mimeType: "image/jpeg",
      dialogTitle: "Поділитися фото природи"
    });

  }
}
```



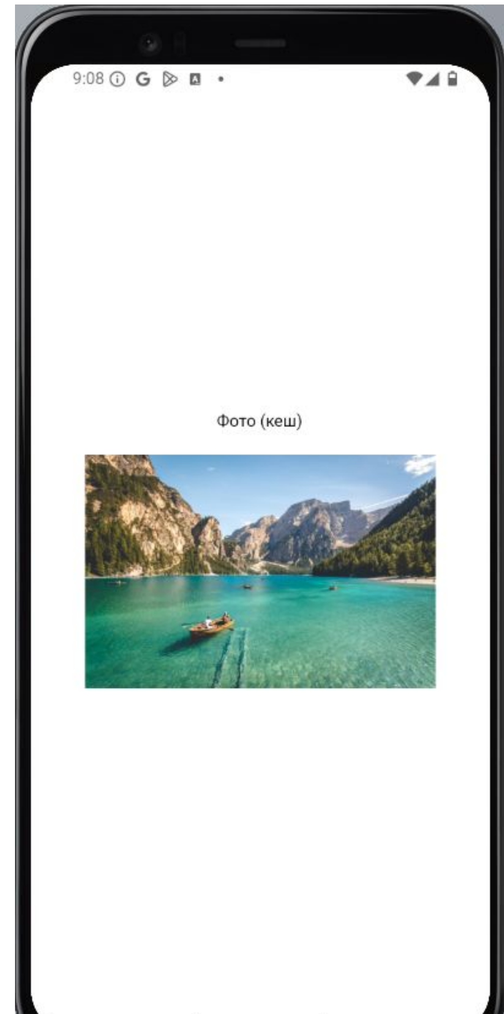
Практичний кейс - Кешування зображень

Замість того, щоб кожного разу завантажувати аватарку користувача з сервера (що витрачає трафік і час), ми напишемо логіку її локального кешування.

Алгоритм:

1. Перевіряємо, чи є аватарка у `Paths.cache`.
2. Якщо є — використовуємо локальний шлях.
3. Якщо немає — завантажуємо з мережі та зберігаємо в `Paths.cache`.

```
async function getCachedNatureImage(url) {

  const cacheDir = new Directory(Paths.cache, "nature");

  if (!cacheDir.exists) {
    cacheDir.create();
  }

  const localImage = new File(cacheDir, "forest.jpg");

  // якщо файл вже у кеші
  if (localImage.exists) {
    console.log("Фото завантажено з кешу");
    return localImage.uri;
  }

  // якщо немає — завантажуємо
  console.log("Завантаження фото з інтернету");

  const response = await fetch(url);
  const buffer = await response.arrayBuffer();

  localImage.write(new Uint8Array(buffer));

  return localImage.uri;
}
```



expo-image-picker

Додаток має доступ лише до власної файлової області (наприклад, `Paths.document`) і не може напряму переглядати галерею користувача або використовувати камеру. Такий підхід є базовим механізмом безпеки мобільних платформ.

Щоб отримати фото або відео від користувача, використовується спеціальний модуль **expo-image-picker**. Він викликає стандартні системні інтерфейси операційної системи для роботи з медіафайлами. У результаті додаток отримує доступ лише до того файлу, який користувач свідомо обрав.

Основні можливості expo-image-picker:

- **Вибір із галереї** – відкриває системний інтерфейс для вибору фотографій або відео з альбому пристрою.
- **Зйомка через камеру** – дозволяє створити нове фото або відео безпосередньо в додатку.
- **Кадрування (редагування)** – можливість обрізати зображення перед передачею у програму.
- **Метадані (EXIF)** – отримання додаткової інформації про файл (дата створення, параметри зйомки тощо).

Таким чином, **expo-image-picker** виступає безпечним посередником між додатком і системними ресурсами пристрою.

Архітектурний принцип роботи

Після вибору зображення модуль **expo-image-picker** створює тимчасову копію файлу у службовій папці додатка — `Paths.cache`. Після цього бібліотека повертає **URI** цього файлу.

Далі додаток може виконати з файлом стандартні операції за допомогою **expo-file-system**, наприклад:

- зберегти файл у постійну директорію (`Paths.document`);
- обробити або перейменувати;
- відправити файл на сервер.

Таким чином, **expo-image-picker** відповідає за безпечне отримання медіафайлу від користувача, а **expo-file-system** — за подальшу роботу з цим файлом у файловій системі додатка.

expo-image-picker TS

55.0.11 • Public • Published a day ago

Readme | Code Beta | 1 Dependency | 233 Dependents | 225 Versions

Expo Image Picker

Provides access to the system's UI for selecting images and videos from the phone's library or taking a photo with the camera.

API documentation

- Documentation for the latest stable release
- Documentation for the main branch

Installation in managed Expo projects

For managed Expo projects, please follow the installation instructions in the API documentation for the latest stable release.

Installation in bare React Native projects

For bare React Native projects, you must ensure that you have installed and configured the `expo` package before continuing.

Add the package to your npm dependencies

```
npx expo install expo-image-picker
```

Configure for Android

Install

```
> npm i expo-image-picker
```

Repository

github.com/expo/expo

Homepage

docs.expo.dev/versions/latest/sdk/imagepicker/

Weekly Downloads

1 295 639

Version	License
55.0.11	MIT

Unpacked Size	Total Files
528 kB	101

Last publish

a day ago

Встановлення

Встановлення

```
npx expo install expo-image-picker
```

Імпорт

```
import * as ImagePicker from 'expo-image-picker';
```

Конфігурація (Config Plugin):

Оскільки використання камери та галереї підпадає під суворі політики конфіденційності (Privacy Policies) Apple та Google, розробник зобов'язаний задекларувати наміри використання цих даних у файлі `app.json`. Без цього додаток не пройде модерацію в маркетплейсах.

Приклад конфігурації (`app.json`):

```json
{
  "expo": {
    "plugins": [
      [
        "expo-image-picker",
        {
          "photosPermission": "The app accesses your photos to let you share them with your friends."
          "colors": {
            "cropToolbarColor": "#000000",
          },
          "dark": {
            "colors": {
              "cropToolbarColor": "#000000",
            }
          }
        }
      ]
    ]
  }
}
```

Життєвий цикл дозволів

Будь-яка взаємодія з апаратним забезпеченням або особистими даними вимагає асинхронного запиту прав доступу. Бібліотека надає два підходи: декларативний (React-хуки) та імперативний (Promise-методи).

Основні інструменти:

- Асинхронні методи: `requestCameraPermissionsAsync()` та `requestMediaLibraryPermissionsAsync()`.

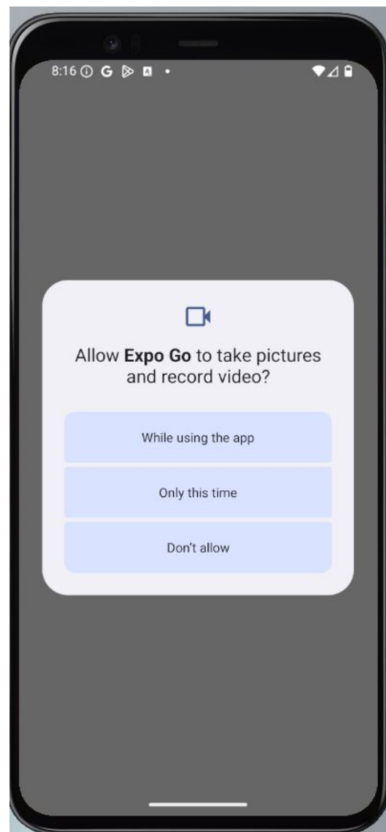
Основні методи виклику інтерфейсів

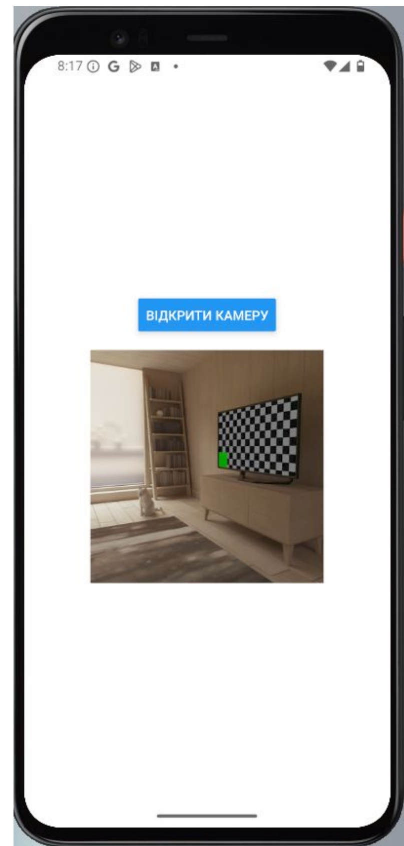
Після отримання прав доступу, додаток ініціює виклик системного UI за допомогою двох головних асинхронних методів. Обидва методи приймають об'єкт конфігурації `ImagePickerOptions`.

Головні методи:

1. `launchImageLibraryAsync(options)` — відкриває файловий менеджер/галерею.
2. `launchCameraAsync(options)` — відкриває видошукач камери.

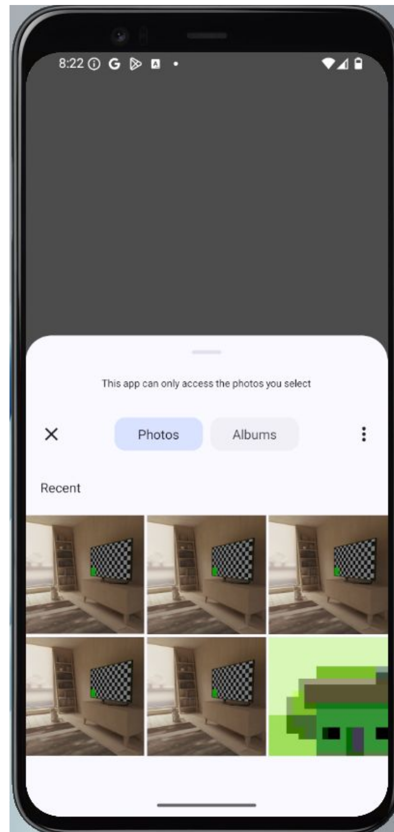
```
const [imageUri, setImageUri] = useState( initialState: null);

const openCamera = async () => {
  // 1. Запит дозволів
  const permission = await ImagePicker.requestCameraPermissionsAsync();

  if (!permission.granted) {
    Alert.alert( title: 'Доступ до камери заборонено');
    return;
  }

  // 2. Запуск камери
  const result = await ImagePicker.launchCameraAsync( options: {
    quality: 1,
  });

  // 3. Якщо користувач зробив фото
  if (!result.canceled) {
    const photoUri = result.assets[0].uri;
    setImageUri(photoUri);
    console.log('Фото URI:', photoUri);
  }
};
```

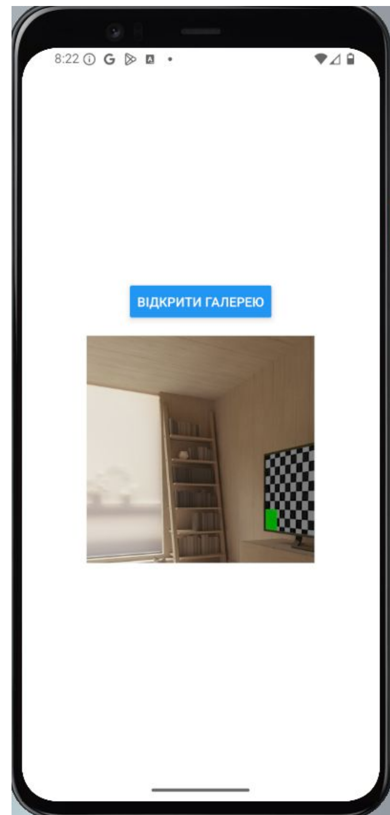





```
LOG  Фото URI: file:///data/user/0/host.exp.exponent/cache/ExperienceData/%2540anonymous%252Flesson_7-f67cebdd-442c-4de0-ae04-c65fd3ba0b2e/ImagePicker/2444467a-f486-41a8-a50c-cf74dad5edd4.jpeg
```

```
const [imageUri, setImageUri] = useState( initialState: null);

const pickImage = async () => {
  // Запит на дозвіл
  const permission = await ImagePicker.requestMediaLibraryPermissionsAsync();

  if (!permission.granted) {
    Alert.alert( title: 'Доступ до галереї заборонено');
    return;
  }

  // Вибір зображення
  const result = await ImagePicker.launchImageLibraryAsync( options: {
    quality: 1,
  });

  if (!result.canceled) {
    const uri = result.assets[0].uri;
    setImageUri(uri);
    console.log('Обрано файл:', uri);
  }
};
```






```
LOG  Фото URI: file:///data/user/0/host.exp.exponent/cache/ExperienceData/%2540anonymous%252Flesson_7-f67cebdd-442c-4de0-ae04-c65fd3ba0b2e/ImagePicker/3965d403-6096-4109-820a-132c42981ead.jpeg
```

Деконструкція результату (**ImagePickerResult**)

Незалежно від обраного методу (камера чи галерея) та кількості файлів, API завжди повертає стандартизований об'єкт `ImagePickerResult`. Його правильний парсинг є критичним для уникнення помилок (Crash) додатку.

Структура об'єкта `ImagePickerResult`: Об'єкт має форму об'єднання типів (Union Type): успішний результат або скасування.

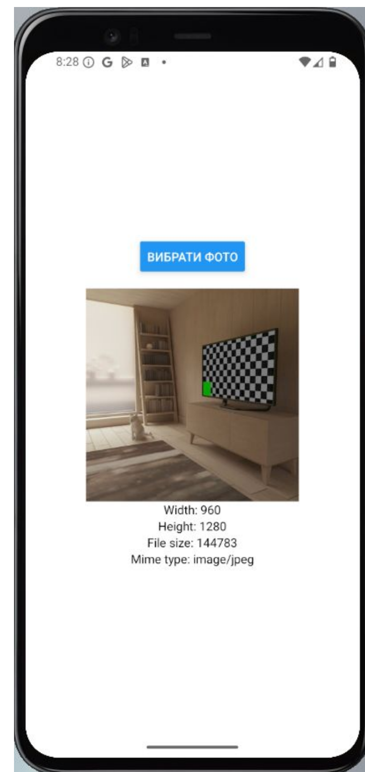
- `canceled` (boolean): Головний прапорець. Якщо `true` — користувач закрив системний UI без вибору файлу, а масив `assets` буде дорівнювати `null`.
- `assets` (Array): Масив об'єктів типу `ImagePickerAsset`.

Ключові властивості `ImagePickerAsset`:

1. `uri` (string): Локальний шлях до тимчасової копії файлу в кеші пристрою (`file:///...`).
2. `width` / `height` (number): Розміри медіа.
3. `fileSize` (number): Розмір файлу в байтах (важливо для обмеження завантажень на сервер).
4. `mimeType` (string): Тип файлу (наприклад, `image/jpeg` або `video/mp4`).
5. `base64` (string | null): Рядок з кодуванням зображення (якщо в опціях передано `base64: true`).
6. `exif` (object | null): Метадані фотографії (якщо передано `exif: true`). Містить GPS-координати, модель камери, дату зйомки.

```
const asset = result.assets[0];

setImageUri(asset.uri);

setInfo({
  width: asset.width,
  height: asset.height,
  size: asset.fileSize,
  type: asset.mimeType
});

console.log("URI:", asset.uri);
console.log("Розмір:", asset.width, "x", asset.height);
console.log("Тип:", asset.mimeType);
console.log("Розмір файлу:", asset.fileSize);
console.log("EXIF:", asset.exif);
```


```
LOG  EXIF: {"ApertureValue": 4, "DateTime": "2026:03:06 20:14:40", "DateTimeDigitized": "2026:03:06 20:14:40", "DateTimeOriginal": "2026:03:06 20:14:40", "ExifVersion": "0220", "ExposureTime": 0.01, "FNumber": 4, "Flash": 0, "FocalLength": 1, "ISOSpeedRatings": 200, "ImageLength": 1280, "ImageWidth": 960, "LightSource": 0, "Make": "Google", "Model": "sdk_gphone64_x86_64", "Orientation": 1, "PixelXDimension": 960, "PixelYDimension": 1280, "ShutterSpeedValue": 6.643, "SubSecTime": "714", "SubSecTimeDigitized": "714", "SubSecTimeOriginal": "714", "WhiteBalance": 0}
```

expo-document-picker

Призначення: Забезпечення доступу до нативного системного інтерфейсу вибору документів (Storage Access Framework на Android та `UIDocumentPickerViewController` на iOS). Дозволяє обирати файли як з локальної пам'яті, так і з хмарних сховищ (iCloud, Google Drive, Dropbox), якщо вони встановлені на пристрої.

Парадокс дозволів: На відміну від камери чи галереї, вам **не потрібно** додавати спеціальні рядки в `app.json` (як-от `NSPhotoLibraryUsageDescription`) чи використовувати React-хуки для запиту дозволів.

Чому так? Безпека гарантується самою ОС (Out-of-Process UI). Додаток не бачить файлової системи — він лише чекає, поки системне вікно закриється і передасть йому посилання на конкретний дозволений файл.

Встановлення

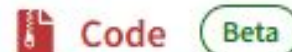
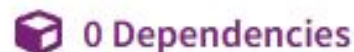
```
npx expo install expo-document-picker
```

Імпорт

```
import * as DocumentPicker from 'expo-document-picker';
```

expo-document-picker TS

55.0.8 • Public • Published 10 days ago

Expo Document Picker

Provides access to the system's UI for selecting documents from the available providers on the user's device.

API documentation

- Documentation for the latest stable release
- Documentation for the main branch

Installation in managed Expo projects

For **managed** Expo projects, please follow the installation instructions in the **API documentation for the latest stable release**.

Installation in bare React Native projects

For bare React Native projects, you must ensure that you have **installed and configured the `expo` package** before continuing.

Install

```
> npm i expo-document-picker
```

Repository

Homepage

docs.expo.dev/versions/latest/sdk/document-picker/

665 825

Version

55.0.8

License

MIT

Unpacked Size

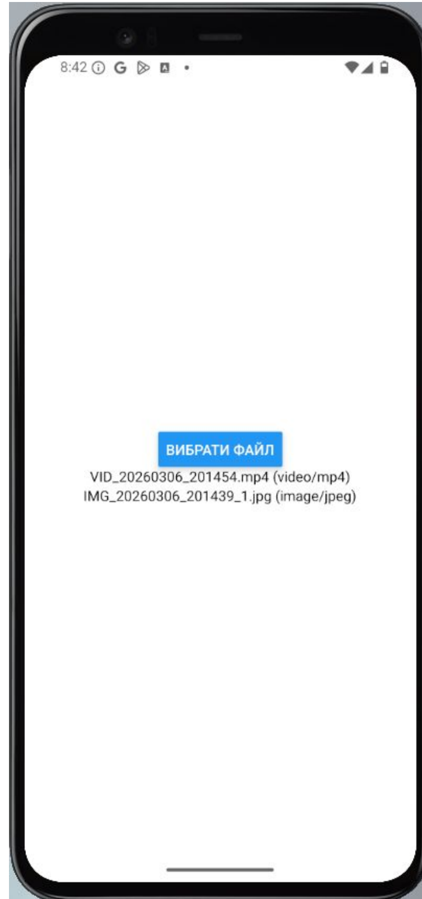
Total Files

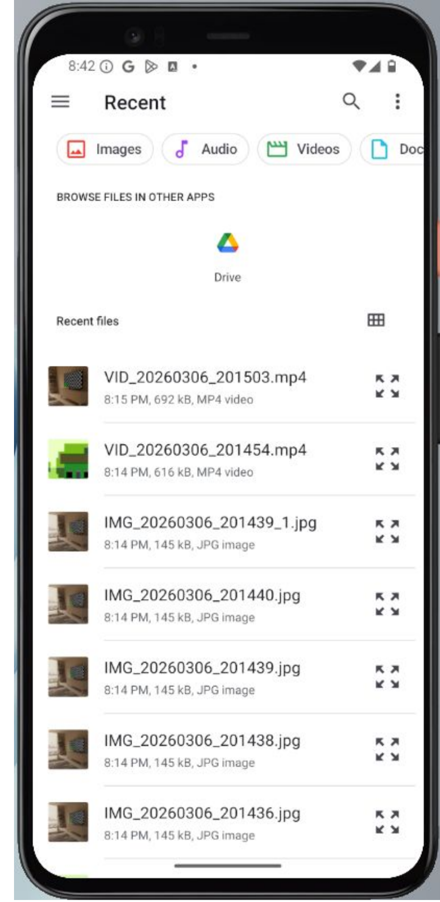
Основний метод та Фільтрація (getDocumentAsync)

Модуль експортує лише один головний метод для виклику інтерфейсу. Його поведінка керується об'єктом параметрів `DocumentPickerOptions`.

- **Синтаксис:** `DocumentPicker.getDocumentAsync(options)`
- **Головний параметр type (MIME-типи):** Замість абстрактних `['images', 'videos']`, тут використовується сувора стандартизація за MIME-типами.
    - `*/*` — будь-який файл (за замовчуванням).
    - `application/pdf` — тільки PDF-документи.
    - `audio/*` — будь-які аудіофайли.
    - Масив типів: `['application/msword', 'application/pdf']` — кілька конкретних форматів.
- **Параметр `multiple`:** Дозволяє вибір кількох файлів одночасно (повертає масив результатів).

```
const [files, setFiles] = useState([]);

Show usages  new *
const pickDocument = async () => {
  const result = await DocumentPicker.getDocumentAsync({
    multiple: true   // дозволити вибір кількох файлів
  });

  if (result.canceled) {
    console.log("Користувач скасував вибір");
    return;
  }

  const selectedFiles = result.assets;
  setFiles(selectedFiles);

  selectedFiles.forEach(file => {
    console.log("Назва:", file.name);
    console.log("URI:", file.uri);
    console.log("Тип:", file.mimeType);
    console.log("Розмір:", file.size);
  });
};
```





Деконструкція результату (DocumentPickerResult)

Після того, як системне вікно закривається, метод повертає об'єкт, архітектурно дуже схожий на результат `image-picker`.

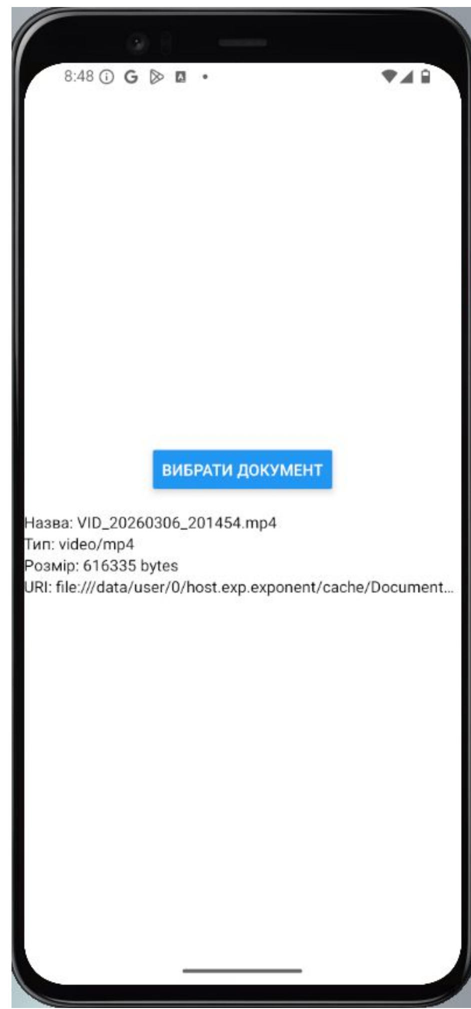
- **Статус скасування:** Поле `canceled` (boolean). Обробка обов'язкова, щоб уникнути помилок, якщо юзер передумав.
- **Масив `assets`:** Містить об'єкти обраних документів.

Властивості об'єкта Asset:

1. `uri` (string): Шлях до файлу.
2. `name` (string): Оригінальна назва файлу (наприклад, `my_resume_2026.pdf`). *На відміну від image-picker, тут ім'я файлу майже завжди зберігається коректно.*
3. `size` (number | undefined): Розмір у байтах (корисно для валідації "файл не більше 5 МБ" перед відправкою на сервер).
4. `mimeType` (string | undefined): Точний MIME-тип обраного файлу.

```
const pickFile = async () => {

  const result = await DocumentPicker.getDocumentAsync({
    multiple: false
  });

  // 1. Перевірка чи користувач скасував вибір
  if (result.canceled) {
    console.log("Користувач закрив файловий picker");
    return;
  }

  // 2. Отримуємо файл із assets
  const file = result.assets[0];

  console.log(result)
```

```
LOG {"assets": [{"lastModified": 1772828098000, "mimeType": "video/mp4", "name": "VID_20260306_201454.mp4", "size": 616335, "uri": "file:///data/user/0/host.exp.exponent/cache/DocumentPicker/ad119644-1edc-4c2a-a981-0facd5f02fe4.mp4"}], "canceled": false}
LOG URI: file:///data/user/0/host.exp.exponent/cache/DocumentPicker/ad119644-1edc-4c2a-a981-0facd5f02fe4.mp4
LOG Назва: VID_20260306_201454.mp4
LOG Розмір: 616335
LOG MIME тип: video/mp4
```

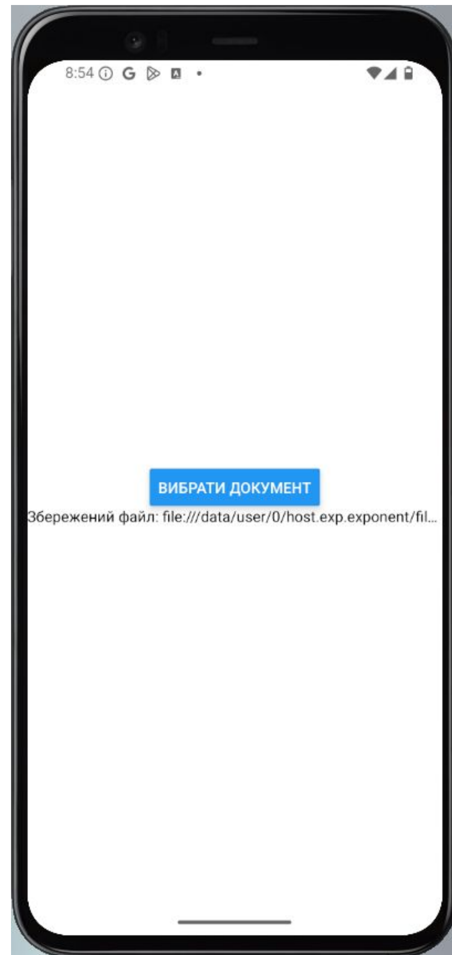


Архітектурний міст — Від Пікера до Постійного Сховища

Головна архітектурна проблема: `expo-image-picker`, і `expo-document-picker` повертають нам шлях (`uri`) до файлу, який лежить у тимчасовій директорії (`Paths.cache`). Якщо користувач закриє додаток, а операційній системі знадобиться вільна пам'ять — вона без попередження видалить ці файли. Якщо ваш додаток покладався на цей `uri` для відображення аватара або збереженого резюме, наступного разу користувач побачить порожній екран або помилку завантаження.

Рішення (Патерн "Перехоплення та Збереження"): Щойно пікер повертає нам успішний результат, ми повинні негайно використати можливості `expo-file-system`, щоб скопіювати або перемістити цей файл із "небезпечної" зони кешу в "безпечну" зону постійного зберігання (`Paths.document`).

```
const asset = result.assets[0];

// файл у cache
const sourceFile = new File(asset.uri);

// файл у постійній директорії
const destinationFile = new File(Paths.document, asset.name);

// копіювання файлу
await sourceFile.copy(destinationFile);

console.log("Файл збережено:", destinationFile.uri);

setSavedPath(destinationFile.uri);
```




```
Import stack:
LOG {"assets": [{"lastModified": 1772828098000, "mimeType": "video/mp4", "name": "VID_20260306_201454.mp4", "size": 616335, "uri": "file:///data/user/0/host.exp.exponent/cache/DocumentPicker/ad119644-1edc-4c2a-a981-0facd5f02fe4.mp4"}], "canceled": false}
LOG URI: file:///data/user/0/host.exp.exponent/cache/DocumentPicker/ad119644-1edc-4c2a-a981-0facd5f02fe4.mp4
LOG Назва: VID_20260306_201454.mp4
LOG Розмір: 616335
LOG MIME тип: video/mp4
LOG Файл збережено: file:///data/user/0/host.exp.exponent/files/VID_20260306_201454.mp4
```