Розробка мобільних додатків

Лекція 3 - Списки

Чому списки - це важливо?

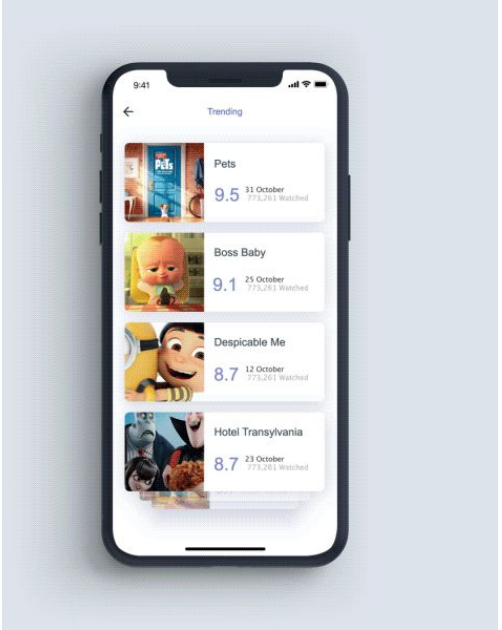
Списки є основним архітектурним патерном відображення контенту в ~90% мобільних додатків.

Ключові сценарії використання:

- **Social Feeds:** Нескінченні стрічки новин (Instagram, Twitter).
- **Chat History:** Історія повідомлень з двонаправленим скролом (Telegram, WhatsApp).
- **Catalogs:** Багатоколонкові сітки товарів (Amazon, Uber Eats).

UX-метрики (User Experience):

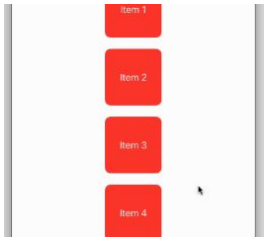
- **Response Time:** Затримка >100ms сприймається користувачем як "лаг".
- **Frame Drops:** Падіння частоти кадрів нижче 55 FPS викликає візуальний дискомфорт.



Типологія списків: Базові структури

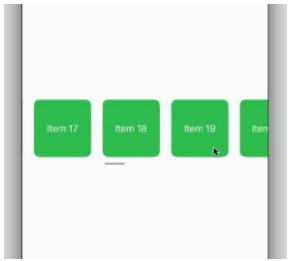
1. Вертикальні списки:

- **Вісь прокрутки:** Y (основна вісь).
- Елементи розташовуються послідовно зверху вниз. Ширина елемента зазвичай дорівнює ширині контейнера.
- *Приклад:* Стрічка налаштувань, стрічка новин.



2. Горизонтальні списки:

- **Вісь прокрутки:** X.
- Елементи розташовуються зліва направо. Часто використовуються як вкладені компоненти ("Каруселі").
- *Приклад:* Stories, банери акцій.



3. Сітки:

- **Структура:** Матриця елементів.
- **Реалізація:** Вертикальний список, де кожен рядок містить кілька елементів (Columns).
- *Приклад:* Галерея фотографій.

Типологія списків: Секційні структури

Списки, що оперують даними, згрупованими за певним критерієм (абетка, дата, категорія).

Архітектурні особливості:

- **Data Structure:** Масив об'єктів, де кожен об'єкт містить заголовок секції та вкладений масив даних (`data: []`).
- **Section Headers:** Компоненти-роздільники, що вставляються між групами даних.
- **Sticky Headers:** Механізм "прилипання" заголовка активної секції до верхньої частини Viewport під час скролу (Native реалізація на iOS/Android).

Застосування:

- Контактні книги (А-Я).
- Меню закладів (Категорії страв).

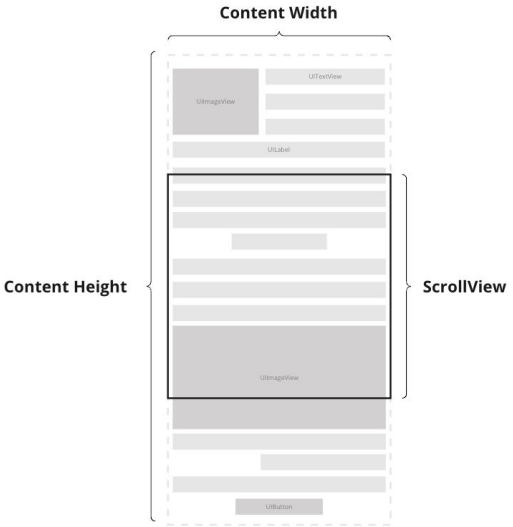
ScrollView

ScrollView - це базовий контейнер для прокрутки, який реалізує принцип **негайного монтування**.

ScrollView · React Native

Принцип роботи:

1. **Initialization:** При старті компонента ініціалізується вся ієрархія вкладених дочірніх еле
2. **Layout Calculation:** Рушій Yoga розраховує координати всіх N елементів.
3. **Rendering:** Створення Native Views для всіх елементів одночасно, незалежно від їх вид



Алгоритмічна складність:

- **Startup Time:** $O(N)$, де N — кількість елементів.
- **Memory Usage:** $O(N)$. Споживання пам'яті зростає лінійно.

ScrollView

Критичні технічні недоліки:

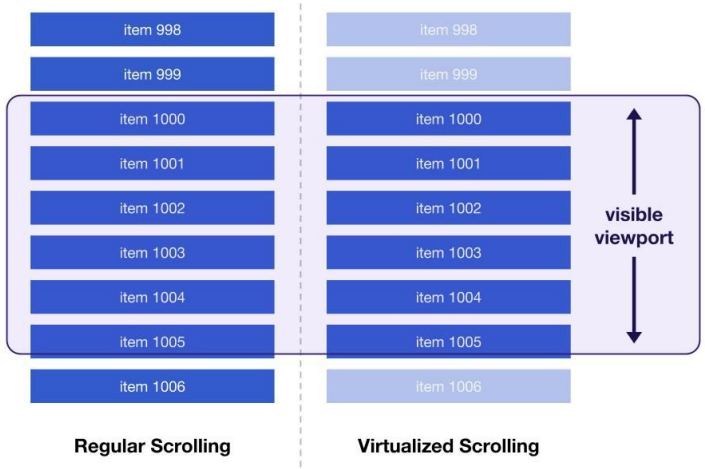
- **Споживання RAM:** Кожен елемент списку (особливо з бітмапами зображень) резервує місце в оперативній пам'яті це призводить до ризику **Out of Memory Crash**.
- **Деградація FPS:** Оскільки всі елементи є частиною активного Native UI Tree, графічний процесор (GPU) та CPU змушені прораховувати геометрію та перемальовувати навіть ті об'єкти, що знаходяться поза межами видимості.
- **Відсутність життєвого циклу "вікна":** Компоненти всередині ScrollView не розмонтовуються, що унеможливлює очищення ресурсів під час скролу.

Концепція віртуалізації

Архітектурний патерн, при якому рендериться лише підмножина елементів, що знаходяться у **Viewport** (видимій частині екрана) плюс невеликий буфер безпеки (Buffer Zone).

Механізм роботи:

1. **Viewable Window:** Обчислення діапазону індексів [start, end], які перетинаються з екраном.
2. **Mounting:** Створення компонентів, що входять у цей діапазон.
3. **Unmounting:** Знищення (або переховування) компонентів, що вийшли за межі буфера.
4. **Spacer Views:** Використання порожніх контейнерів зверху і знизу списку для емуляції повної висоти контенту (щоб скрол-бар поводився коректно).



Результат:

Споживання пам'яті стає константним $O(1)$ (відносно розміру вікна), а не лінійним $O(N)$.

FlatList

FlatList — це стандартна реалізація віртуалізованого списку в React Native.

Архітектура:

- Базується на компоненті **`VirtualizedList`**.

Ключові характеристики:

- **Lazy Rendering:** Елементи створюються лише за потреби.
- **Unmount Off-screen:** Елементи, що виходять далеко за межі екрана, повністю видаляються з дерева компонентів.
- **Scroll Awareness:** Вбудована підтримка подій `onEndReached` (для пагінації) та `onRefresh` (Pull-to-refresh).

FlatList API:

Для мінімальної роботи компонента необхідні три пропси:

1. **`data` (Array):** Масив даних. Для коректної роботи `PureComponent` всередині списку, посилання на масив має змінюватися лише при реальних змінах даних.
2. **`renderItem` (Function):** Сигнатура: `({ item, index, separators }) => JSX.Element`. Функція, що повертає React Element для конкретного рядка даних.
3. **`keyExtractor` (Function):** Сигнатура: `(item, index) => string`. Повертає унікальний стабільний ідентифікатор для кожного елемента.

```
<FlatList
    data={contacts} // Передаємо масив
    renderItem={renderContact} // Функція для відображення
    keyExtractor={(item) => item.id} // Унікальний ключ
    contentContainerStyle={styles.container}
/>
```

Роль **keyExtractor**

Проблема: React використовує ключі для алгоритму **Diffing** (пошук різниці між старим і новим Virtual DOM). Без стабільних ключів React не може ефективно визначити, чи був елемент переміщений, змінений або видалений.

Анти-паттерн: Використання Index:

```
keyExtractor={(item, index) => index.toString()} // ПОГАНО
```

Якщо ви видалите елемент з початку списку (`index 0`), всі наступні елементи змінять свої індекси. React вирішить, що змінилися *всі* компоненти, і виконає повний ре-рендер всього списку.

Best Practice: Завжди використовуйте унікальні ID сутності з бази даних (`id`, `uuid`, `slug`).

Декорування: ItemSeparatorComponent

Призначення: Рендеринг роздільників між елементами списку без необхідності додавати `marginBottom` до кожного елемента.

Технічні переваги:

- **Інтелектуальна вставка:** Компонент автоматично вставляється *між* елементами, але ігнорується перед першим та після останнього елемента.
- **Layout Optimization:** Усуває необхідність логічних перевірок типу `index === lastIndex` всередині `renderItem`, спрощуючи код картки.

```
<FlatList
    data={data}
    renderItem={renderItem}
    keyExtractor={item => item.id}
    ItemSeparatorComponent={MySeparator}
/>
```

ListEmptyComponent

Призначення: Відображення альтернативного UI, коли масив `data` порожній (`[ ]`).

Сценарії використання:

- Повідомлення "Немає результатів пошуку".
- Заклик до дії ("Створіть свій перший пост").
- Помилки завантаження даних.

Архітектурна особливість: Цей компонент рендериться всередині контейнера прокрутки. Щоб він розтягувався на весь екран, необхідно задати `contentContainerStyle={{ flexGrow: 1 }}` для самого FlatList.

```
<View style={{ flex: 1 }}>
    <FlatList
        data={items}
        renderItem={renderItem}
        keyExtractor={(item) => item.id}
        ListEmptyComponent={
            <View style={styles.emptyContainer}>
                <Text>Список порожній</Text>
            </View>
        }
        contentContainerStyle={{ flexGrow: 1 }}
    />
</View>
```

Хедери та Футери

Props: `ListHeaderComponent` та `ListFooterComponent`.

Призначення: Рендеринг контенту, який скролиться разом зі списком, але не є частиною масиву даних.

Типові кейси:

- **Header:** Рядок пошуку (Search Bar), великі заголовки, банери.
- **Footer:** Індикатор завантаження (ActivityIndicator) для нескінченного скролу (Pagination).

Перевага над зовнішнім ScrollView: Дозволяє уникнути помилки вкладеності (*"VirtualizedList should never be nested inside plain ScrollView"*), зберігаючи єдиний контекст прокрутки та оптимізацію пам'яті.

ListHeaderComponent, ListFooterComponent

```
<FlatList
    data={DATA} // Масив даних
    ListHeaderComponent={<Text style={styles.header}>Header</Text>}
    ListFooterComponent={<Text style={styles.footer}>Footer</Text>}
    keyExtractor={(item) => item.id} // Унікальний ключ для кожного елемента
    renderItem={({ item }) => ( // Рендер кожного елемента списку
        <View style={styles.item}>
            <Text style={styles.text}>{item.title}</Text>
        </View>
    )}
/>
```

Стилізація: **style vs contentContainerStyle**

1. prop `style`:

- Застосовується до зовнішньої оболонки (Wrapper) ScrollView.
- Визначає розмір та позицію самого "вікна" на екрані.
- *Використання:* Задати ширину/висоту списку, `backgroundColor` фону.

2. prop `contentContainerStyle`:

- Застосовується до внутрішнього контейнера, який охоплює всі елементи (Items + Header + Footer).
- *Використання:* `padding` (відступи контенту від країв), вирівнювання елементів (наприклад, центрування порожнього компонента).

Горизонтальний режим

Активація: `horizontal={true}`.

Зміни в Layout Engine:

- Flex-контейнер перемикається в режим `flexDirection: 'row'`.
- Всі елементи шикуються зліва направо.

Особливості UI/UX:

- **`showsHorizontalScrollIndicator={false}`:** Рекомендується приховувати скрол-бар для чистоти дизайну.
- **Вкладеність:** Горизонтальні списки часто вкладаються у вертикальні (патерн "Carousels inside Feed"). Це легально і працює стабільно, оскільки напрямки скролу перпендикулярні (Orthogonal Scrolling).

Багатоколонковий режим (Grids)

Props: `numColumns` (number).

Функціонал: Автоматично розбиває простір по горизонталі на вказану кількість колонок.

- Якщо `numColumns > 1`, `FlatList` ігнорує налаштування `flexDirection` у елементів і примусово вибудовує їх у рядки.

Обмеження:

- Не підтримується разом з `horizontal={true}`.
- Вимагає, щоб всі елементи мали однаковий розмір (або контрольовану висоту).

```
<FlatList
    data={data}
    renderItem={renderItem}
    keyExtractor={(item) => item.id} //
    numColumns={numColumns}
    contentContainerStyle={styles.listContent}
/>
```

Нескінченний скрол

Механізм: Патерн завантаження даних порціями (Pagination), коли користувач наближається до кінця списку.

Prop: `onEndReached`. Функція, яка викликається один раз, коли позиція скролу перетинає порогове значення.

Типова логіка обробника:

1. Перевірка: чи вже йде завантаження? (Is Loading?)
2. Перевірка: чи є ще дані на сервері? (Has More?)
3. API запит за наступною сторінкою (`page + 1`).
4. Додавання нових даних до існуючого масиву (`[...oldData, ...newData]`).

Ризики: Множинні виклики функції, якщо користувач швидко скролить "туди-сюди" в зоні порогу. Вимагає захисту (Debounce або прапор `isLoading`).

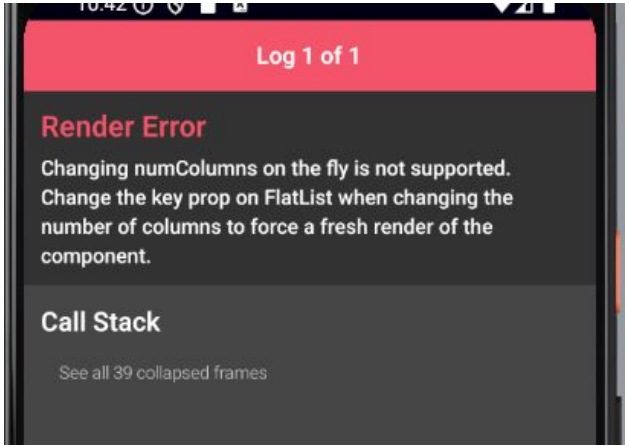
Динамічна зміна кількості колонок

Проблема: Зміна пропса `numColumns` (наприклад, з 2 на 3 при повороті екрана) викликає критичну помилку React Native, оскільки це змінює структуру вузлів VirtualizedList.

Рішення: Щоб змінити кількість колонок, необхідно змінити проп `key` самого компонента FlatList. Це змушує React повністю знищити старий список і створити новий з новою конфігурацією.

Реалізація:

```
<FlatList
    // РІШЕННЯ: Зміна key змушує React перестворити компонент без помилок
    key={columns}
    data={data}
    numColumns={columns}
    keyExtractor={(item) => item.id} //
    contentContainerStyle={styles.listContent} //
    renderItem={({ item }) => (
        <View style={[styles.item, { height: columns === 2 ? 150 : 100 }]}>
            <Text style={styles.itemText}>{item.name}</Text>
        </View>
    )}
/>
```



Налаштування порогу пагінації

Prop: `onEndReachedThreshold` (number).

Значення: Число від `0` до N, яке означає відстань від кінця списку у **висотах екрану (Screen Heights)**.

- `0`: Викликати функцію, тільки коли піксель останнього елемента з'явився на екрані.
- `0.5`: Викликати, коли до кінця залишилось пів екрану.
- 2: Викликати, коли до кінця ще 2 екрани скролу.

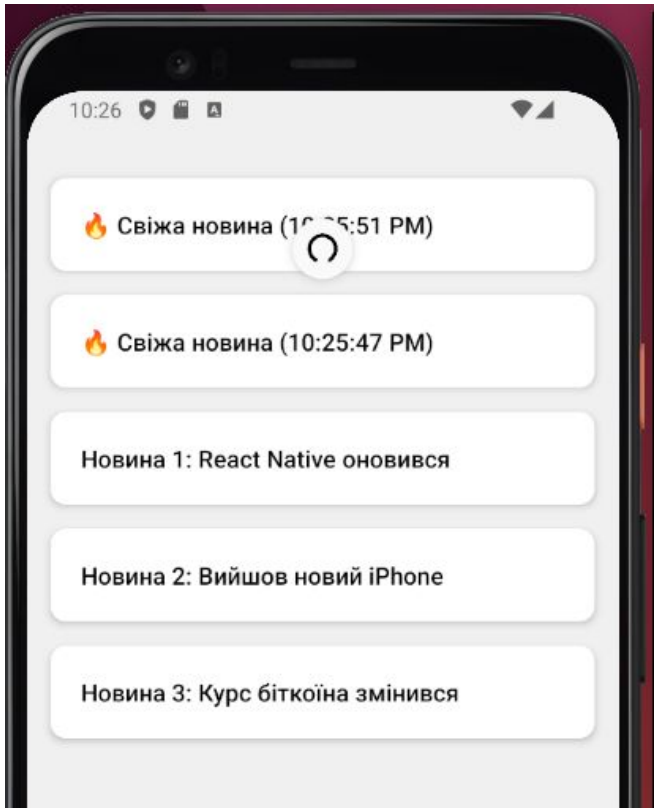
Рекомендація: Для плавного UX (щоб користувач не бачив спінера) використовуйте значення `0.5` – 1. Це дає час завантажити дані до того, як користувач дійде до низу.

Pull-to-Refresh (Оновлення потягуванням)

Стандартний патерн: Оновлення списку шляхом потягування його вниз з верхньої точки.

Props:

1. **`onRefresh` (function):** Функція, що викликається при виконанні жесту. Зазвичай робить скидання пагінації (`page = 1`) і новий запит API.
2. **`refreshing` (boolean):** Стан, що контролює видимість спінера оновлення.
   - `true`: показує спінер.
   - `false`: ховає спінер.



Інвертований режим (Chat Mode)

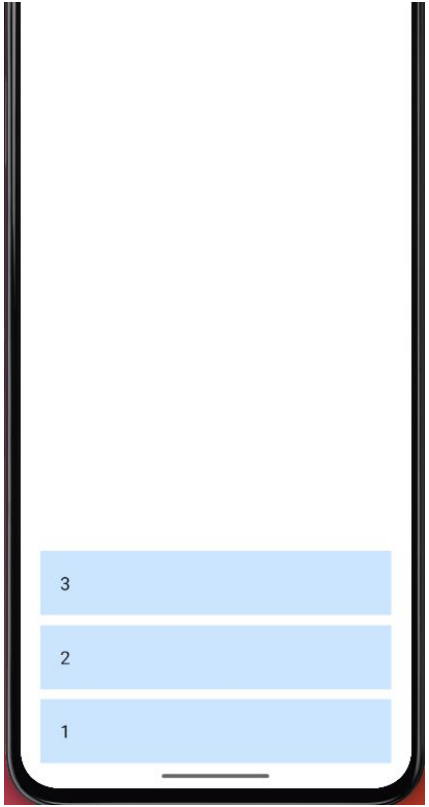
Prop: `inverted={true}`.

Застосування: Критично необхідний для створення інтерфейсів чатів (Messages, WhatsApp), де контент починається знизу екрана, а нові повідомлення додаються "знизу".

Технічна реалізація:

- React Native застосовує трансформацію `scaleY: -1` до контейнера списку, перевертаючи його.
- Одночасно застосовується `scaleY: -1` до кожного дочірнього елемента, щоб повернути їм нормальну орієнтацію.
- **Результат:** Перший елемент масиву (`index: 0`) візуально знаходиться в самому низу екрана.

При використанні `inverted`, напрямок скролу та логіка `onEndReached` також інвертуються (тригер спрацьовує при скролі вгору).



Програмний скрол: Робота з **Ref**

Концепція: Для маніпуляцій зі списком без участі користувача (наприклад, кнопка "Вгору" або перехід до коментаря) ми використовуємо **Ref** (посилання на екземпляр компонента).

Ініціалізація:

```
const flatListRef = useRef(null);

<FlatList
  ref={flatListRef}
  data={data}
  renderItem={renderItem}
/>
```

Доступні методи: Через `flatListRef.current` ми отримуємо доступ до імперативних методів керування: `scrollToIndex`, `scrollToItem`, `scrollToOffset`, `scrollToEnd`.

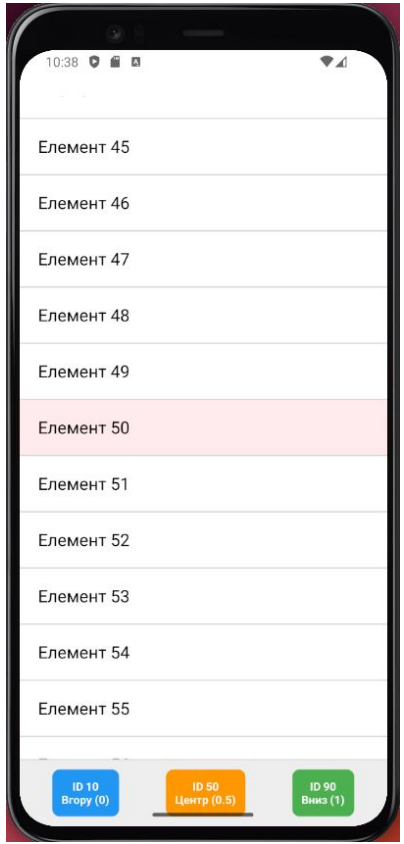
Навігація до елемента: scrollToIndex

Метод: `scrollToIndex(params)`.

Параметри:

- `index` (number): Індекс цільового елемента в масиві `data`.
- `animated` (boolean): `true` для плавної анімації, `false` для миттєвого стрибка.
- `viewPosition` (number): Де має опинитися елемент після скролу:
  - `0`: Вгорі екрана.
  - `0.5`: По центру екрана.
  - `1`: Внизу екрана.

```
const goToIndex = (index) => {
  flatListRef.current?.scrollToIndex({
    index,
    animated: true,
    viewPosition: 0.5
  });
};
```



Проблема **scrollToIndex** та **onScrollToIndexFailed**

Технічне обмеження: `scrollToIndex` гарантовано працює, тільки якщо ви використовуєте `getItemLayout` (де висота елементів фіксована або відома). Якщо висота динамічна, FlatList не знає координати елемента, який ще не був відрендерений (знаходиться далеко за межами Viewport), і викидає помилку.

Обробка помилки: Потрібно використовувати колбек `onScrollToIndexFailed`.

Суть: Скролимо до приблизної позиції, чекаємо рендеру, коригуємо скрол.

```
onScrollToIndexFailed={(info) => {
  console.log('⚠️ Елемента немає в пам'яті. Виконуємо план Б...');

  const wait = new Promise( executor: resolve => setTimeout(resolve, 500));

  // Крок 1: Скролимо до ПРИБЛИЗНОЇ позиції
  // info.averageItemLength — це середня висота вже відрендерених елементів
  const offset = info.index * info.averageItemLength;

  flatListRef.current?.scrollToOffset( params: {
    offset: offset,
    animated: true,
  });

  // Крок 2: Чекаємо рендеру і коригуємо (Retry)
  wait.then(() => {
    flatListRef.current?.scrollToIndex( params: {
      index: info.index,
      animated: true,
      viewPosition: 0.5,
    });
  });
}}
```

Кнопка "Вгору" та **scrollToEnd**

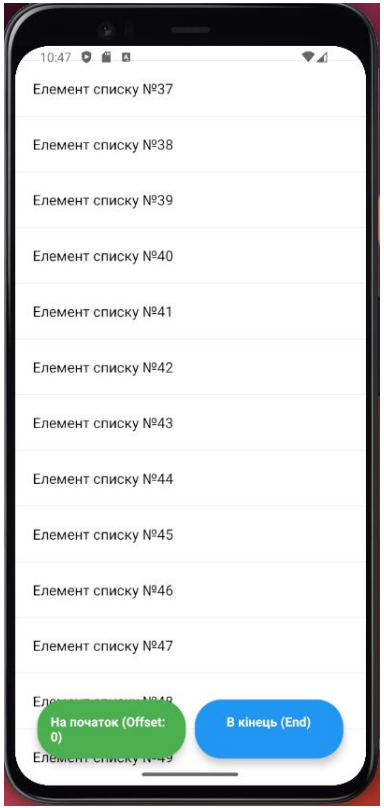
Метод `scrollToOffset`: Використовується для скролу до конкретної піксельної координати. Найчастіший кейс — повернення на початок.

```
// Scroll to Top
flatListRef.current?.scrollToOffset({ offset: 0, animated: true });
```

Метод `scrollToEnd`: Скролить до кінця контенту. Корисно в чатах при відправці нового повідомлення.

```
flatListRef.current?.scrollToEnd({ animated: true });
```

Примітка: У поєднанні з `inverted={true}`, `scrollToEnd` відправить вас візуально вгору (до найстаріших повідомлень), оскільки вісь Y перевернута.



Відстеження видимості (Viewability)

Prop: `onViewableItemsChanged`.

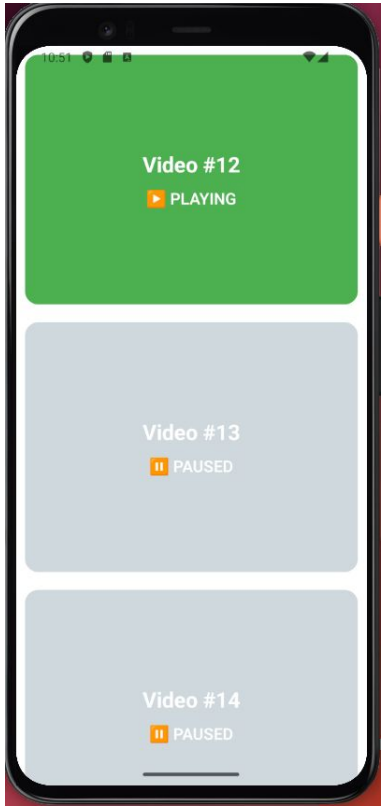
Призначення: Асинхронний колбек, який повідомляє, які елементи зараз знаходяться у Viewport. Це фундаментальний механізм для реалізації:

- **Аналітики:** Відправка події "Impression" (користувач побачив рекламу).
- **Медіа:** Автоматичний запуск відео (Autoplay), коли воно потрапляє в центр екрана.

Сигнатура:

```
const onViewableItemsChanged = ({ viewableItems, changed }) => {
  // viewableItems: масив об'єктів, які зараз видимі
  // changed: масив об'єктів, статус видимості яких змінився
};
```

Важливо: Ця функція **має бути стабільною**. Її потрібно створювати через `useCallback` або оголошувати змінну поза компонентом. Якщо посилання на функцію змінюється при кожному рендері, `FlatList` працюватиме некоректно



Конфігурація видимості (ViewabilityConfig)

Prop: `viewabilityConfig`.

Призначення: Визначає критерії, за яких елемент вважається "видимим".

Параметри конфігурації:

1. **`itemVisiblePercentThreshold` (number):** Відсоток площі елемента, який має бути у Viewport.
   - `50`: Елемент видимий, якщо показано хоча б половину.
   - `100`: Елемент має бути видимим повністю.
2. **`minimumViewTime` (number):** Час у мілісекундах, протягом якого елемент має залишатися у зоні видимості, щоб подія спрацювала (фільтрація швидкого скролу).
3. **`waitForInteraction` (boolean):** Чи чекати взаємодії користувача перед реєстрацією переглядів.

```
const viewConfigRef = useRef({ itemVisiblePercentThreshold: 50 });
// Передаємо у FlatList саме viewConfigRef.current
```

Первинний рендеринг: initialNumToRender

Вплив на продуктивність: Цей проп визначає кількість елементів, які рендеряться **синхронно** в першому кадрі.

Проблема:

- **Замале значення:** Користувач бачить порожнє місце знизу екрана.
- **Завелике значення:** Збільшується час до інтерактивності (TTI), навігація на екран "завмирає".

Формула розрахунку:

$$N = \lceil \frac{ScreenHeight}{ItemHeight} \rceil$$

Тобто, рівно стільки елементів, скільки потрібно, щоб заповнити екран, плюс один-два про запас.

Default: `10`. Якщо у вас складні картки, зменште це число.

Проблема вимірювання

Як працює FlatList за замовчуванням: Щоб намалювати скрол-бар і знати, куди переміститися при виклику `scrollToIndex`, FlatList мусить знати точні координати (x, y) та розміри (width, height) кожного елемента. Але він їх не знає, доки не відрендерить елемент.

Наслідки:

1. **Jumping Scroll:** Скрол-бар "стрибає", коли підвантажуються нові елементи, змінюючи загальну висоту контенту.
2. **Неможливість ScrollToIndex:** Ви не можете проскролити до 500-го елемента, якщо елементи 0–499 ще не були відрендерені (система не знає, де саме починається 500-й елемент у пікселях).
3. **Асинхронна затримка:** Витрачається час CPU на подію `onLayout`.

Рішення: **getItemLayout**

Суть методу: Ми даємо FlatList формулу, за якою він може **миттєво** розрахувати позицію будь-якого елемента, не рендерячи його. Ми обходимо систему Layout Engine (Yoga).

Умова використання: Ви повинні знати висоту своїх елементів наперед. Найкращий сценарій — **фіксована висота** для всіх рядків.

Аргументи функції: React Native очікує, що ви повернете об'єкт з трьома полями:

- `length`: Висота (або ширина) поточного елемента.
- `offset`: Відстань від початку списку до верхньої межі поточного елемента (сума висот усіх попередніх елементів).
- `index`: Індекс поточного елемента.

Реалізація для фіксованої висоти

Це найпоширеніший і найефективніший сценарій. Якщо всі картки мають висоту `100px`.

Формула:

$$Offset = Index \times Height$$

Результат: Тепер `scrollToIndex({ index: 5000 })` спрацює миттєво і з ідеальною точністю, навіть якщо список ще не завантажився.

```
const ITEM_HEIGHT = 100; // Висота картки + відступи

<FlatList
  data={data}
  renderItem={renderItem}
  getItemLayout={(data, index) => ({
    length: ITEM_HEIGHT,
    offset: ITEM_HEIGHT * index,
    index,
  })}
/>
```

SectionList

Визначення: `SectionList` — це спеціалізований компонент-обгортка над `VirtualizedList`, призначений для відображення **згрупованих** наборів даних.

Коли використовувати: Якщо ваші дані мають ієрархію "Категорія -> Елементи", використання `FlatList` призведе до складних маніпуляцій з індексами. `SectionList` вирішує це "з коробки".

Типові приклади:

- **Контакти:** Групування за алфавітом (А, Б, В...).
- **Меню ресторану:** Групування за типом страв (Закуски, Основні, Напої).
- **Історія транзакцій:** Групування за датами (Сьогодні, Вчора, Минулий тиждень).

Специфічна структура даних

Відмінність від FlatList: Замість пропу `data` (плаский масив), `SectionList` приймає проп **`sections`**.

Формат даних: Це масив об'єктів, де кожен об'єкт описує одну секцію. Обов'язковим є лише ключ `data` (масив елементів секції). Інші ключі (наприклад, `title`, `id`) є довільними.

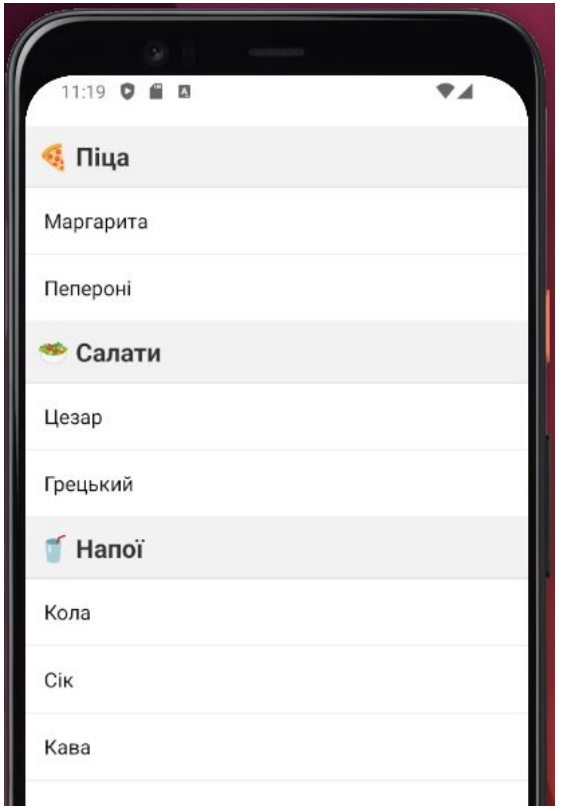
```
const MENU_ITEMS = [
  {
    title: '🍕 Піца', // Назва секції
    data: ['Маргарита', 'Пепероні']
  },
  {
    title: '🥗 Салати',
    data: ['Цезар', 'Грецький' ]
  },
  {
    title: '🥤 Напої',
    data: ['Кола', 'Сік', 'Кава']
  }
];
```

```
<SectionList
  sections={MENU_ITEMS}
```

Рендеринг заголовків секцій

Prop: `renderSectionHeader`.
Функціонал: Функція, яка відповідає за візуалізацію хедера кожної групи. Вона отримує об'єкт секції як аргумент.
Особливість: Цей компонент вставляється в потік списку перед першим елементом масиву `data` відповідної секції.

```
renderSectionHeader={({ section }) => (
  <View style={styles.header}>
    <Text style={styles.headerText}>{section.title}</Text>
  </View>
)}
```



Секційний Футер

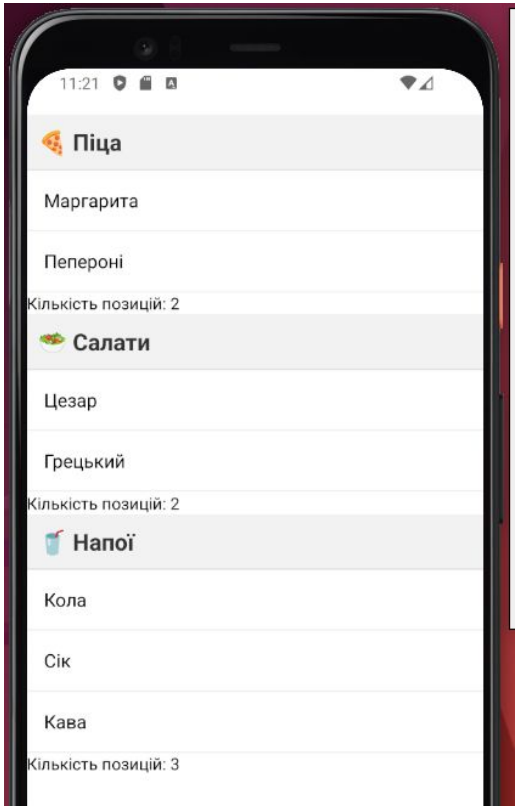
Prop: `renderSectionFooter`.

Поведінка: Функція викликається для **кожної** групи після рендерингу всіх її елементів, але перед наступним заголовком.

```
renderSectionFooter={({ section }) => (
  <View style={styles.sectionFooter}>
    <Text>Всього в категорії: {section.data.length}</Text>
  </View>
)}
```

Використання:

- Підсумки по групі (сума чеку в категорії).
- Кнопка "Дивитись всі" (якщо показано лише топ-3 елементи).
- Візуальне закриття блоку.



Глобальні Хедер та Футер **(List Header/Footer)**

Props: `ListHeaderComponent` / `ListFooterComponent`.

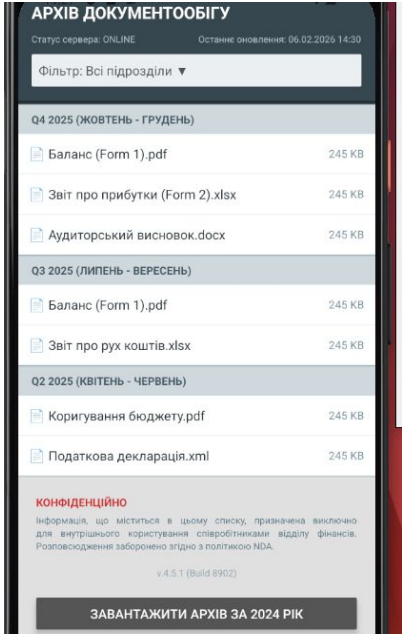
Поведінка: Це компоненти, які рендеряться **один раз** для всього списку.

- **ListHeader:** Найперший елемент списку (над першою секцією).
- **ListFooter:** Найостанніший елемент списку (під останньою секцією).

Використання:

- *Header:* Глобальний пошук, рекламний банер, загальний заголовок сторінки.
- *Footer:* Індикатор завантаження ("Loading more..."), копірайт, відступи внизу екрана.

Важливо: Вони скроляться разом з усім контентом.



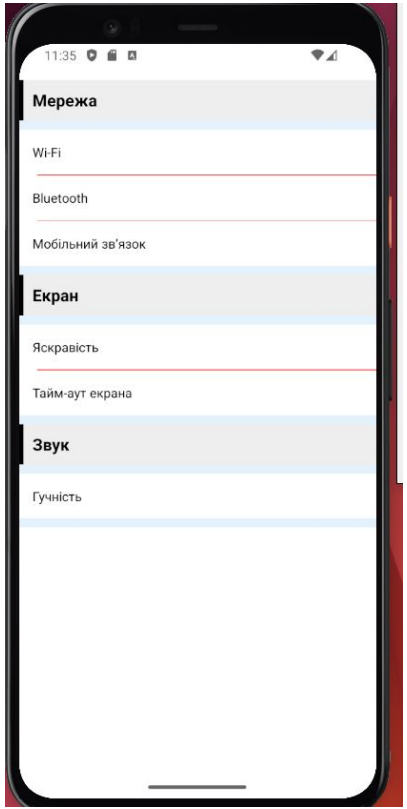
Роздільники (Separators) в SectionList

У `SectionList` є два рівні роздільників:

1. `ItemSeparatorComponent`: Працює так само, як у FlatList. Вставляється **між елементами** всередині секції.

- Не вставляється між останнім елементом секції та футером секції.

2. `SectionSeparatorComponent`: Вставляється **між секціями** (тобто між футером попередньої секції та хедером наступної), а також між хедером секції та першим елементом.



contentContainerStyle

`contentContainerStyle` використовується для задання **внутрішніх відступів усього контенту списку**, а не окремих елементів.

Він застосовується **до контейнера з елементами SectionList**, включно з:

- заголовками секцій ;
- футерами секцій.

```
const insets = useSafeAreaInsets();

<SectionList
  contentContainerStyle={{
    paddingHorizontal: 16,
    paddingBottom: insets.bottom + 20
  }}
  // ...
/>
```

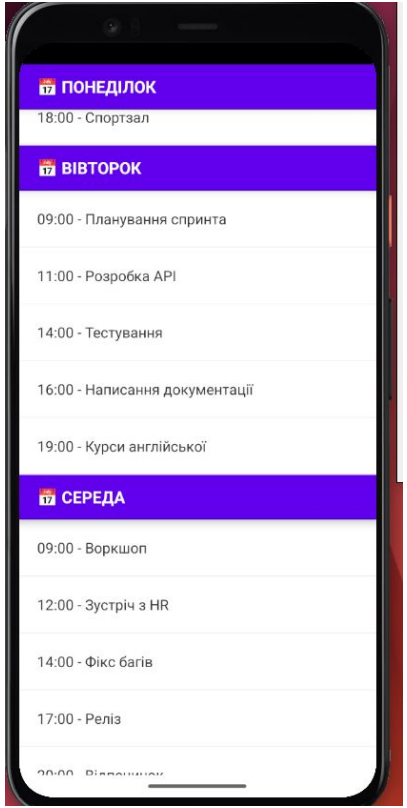
Sticky Headers

Prop: `stickySectionHeadersEnabled`.

Поведінка: Коли користувач скролить список, заголовок поточної групи "прилипає" до верхньої частини екрана і перекриває контент, що прокручується під ним. Він залишається там доти, доки заголовок *наступної* секції не "виштовхне" його.

Відмінності:

- **iOS:** Увімкнено за замовчуванням (`true`). Це стандартна поведінка `UITableView`.
- **Android:** Вимкнено за замовчуванням (`false`).
- **Рекомендація:** Для кращого UX явно встановлювати `true` для обох платформ.



Індексація та **keyExtractor** у **SectionList**

Складність ключів: Оскільки у SectionList є групи та елементи, стратегія ключів трохи змінюється.

keyExtractor: Отримує `item` (елемент даних) та `index` (індекс всередині секції). Повинен повертати унікальний ключ **в межах всього списку**, а не тільки секції.

Приклад помилки: Якщо у вас є "Яблуко" в секції "Фрукти" і "Яблуко" в секції "Покупки", і ви використовуєте назву як ключ — буде колізія.

Рішення: Комбінуйте ID секції та ID елемента, якщо вони не унікальні глобально.

```
keyExtractor={(item, index) => item.id + index}
```

Специфічні методи **SectionList**

Як і `FlatList`, цей компонент має набір імперативних методів (через `ref`), але з адаптацією під структуру.

`scrollToLocation(params)`

Параметри:

- `sectionIndex` (number): Індекс групи (наприклад, 2-га категорія).
- `itemIndex` (number): Індекс елемента всередині цієї групи (наприклад, 5-й товар).
- `viewOffset` (number): Зміщення в пікселях (корисно для фіксації під липким хедером).

```
const go = () => {
  listRef.current.scrollToLocation({
    sectionIndex: 1,
    itemIndex: 3,
  });
};
```

Підводні камені **SectionList**

1. onEndReached: Працює так само, як у `FlatList`, але додавання даних у кінець масиву секцій (нова секція) працює добре, а от додавання елементів у *існуючу* секцію може викликати стрибки, якщо не збережено посилання на об'єкт секції.

2. Мерехтіння хедерів: На Android при використанні складних компонентів у `renderSectionHeader` може виникати візуальний лаг при "прилипанні". *Хедери повинні бути* максимально легкими (простий `<Text>` і `<View>` з фіксованою висотою).

3. Відсутність `numColumns`: `SectionList` **не підтримує** сітку (Grid layout). Для цього потрібно писати кастомну логіку всередині `renderItem` (рендерити по 2 елементи в рядку вручну).

VirtualizedList

Ієрархія: У React Native `VirtualizedList` — це базовий компонент ("двигун"), на якому побудовані всі інші списки.

- **`FlatList`** — це обгортка над `VirtualizedList`, налаштована для роботи зі звичайними масивами (`Array`).
- **`SectionList`** — це теж обгортка над `VirtualizedList`, яка вміє трансформувати структуру секцій у лінійний список індексів.

Призначення: Він реалізує всю складну логіку віртуалізації:

1. Відстеження скролу.
2. Визначення вікна видимості.
3. Mount/Unmount комірок для економії пам'яті.
4. Асинхронний рендеринг батчами.

Висновок: Коли ви використовуєте `FlatList`, ви *де-факто* використовуєте `VirtualizedList`

Обмеження **FlatList:**

FlatList прив'язаний до типу даних Array. Якщо ваші дані зберігаються в Map, Set або Immutable.js List, доводиться робити конвертацію: Array.from(mySet).*Ціна:* $O(N)$ операцій процесора + подвоєння використання пам'яті (дублювання даних).

Філософія VirtualizedList:

Цей компонент **агностичний** до структури даних. Він не знає, що таке "масив". Замість передачі даних у зрозумілому форматі, ви передаєте йому **Інструкцію (Протокол)**, як читати ваші дані.Це реалізується через принцип **Inversion of Control**.

Prop: getItemCount

Оскільки `VirtualizedList` є агностичним до типу даних, він не робить припущень щодо їх структури. Він не звертається напряму до властивості `.length`, оскільки дані можуть зберігатися у вигляді масивів, зв'язних списків, хеш-таблиць (`Map`), наборів (`Set`)

Метод дозволяє рушію віртуалізації отримати загальну кількість елементів для:

1. Розрахунку повної висоти контенту (Content Height).
2. Визначення діапазону індексів для рендерингу.
3. Коректного відображення індикатора прокрутки.

Prop: getItem

Питання списку:

"Мені потрібно відрендерити рядок №5. Дай мне дані саме для цього рядка."

Логіка:

Це функція доступу. Вона викликається системою віртуалізації "ліниво" — тільки тоді, коли рядок наближається до екрана.

Приклади:

- Для Масиву: return data[index];

Критично важливо:

Функція getItem має бути максимально швидкою, оскільки вона викликається сотні разів під час швидкого скролу.

Реконструкція **FlatList**

Доведемо, що `FlatList` — це просто синтаксичний цукор, відтворивши його функціонал вручну через `VirtualizedList`.

Цей код працює ідентично до стандартного `FlatList`. Саме це відбувається "під капотом" бібліотеки React Native.

```
import { VirtualizedList } from 'react-native';

const MyFlatList = ({ data, renderItem }) => {
  return (
    <VirtualizedList
      data={data} // Передаємо масив як є

      // 1. Вчимо список рахувати довжину масиву
      getItemCount={(items) => items.length}

      // 2. Вчимо список діставати елемент за індексом
      getItem={(items, index) => items[index]}

      renderItem={renderItem}
      keyExtractor={(item) => item.id}
    />
  );
};
```

Проблема стабільності посилань

Критичне правило: Функція `getItem` має повертати **те саме посилання** для тих самих вхідних даних, якщо це можливо.

```
// ❌ НЕПРАВИЛЬНО
getItem={(data, index) => ({ ...data[index], extra: true })}
```

Тут ми створюємо *новий об'єкт* при кожному виклику `getItem`. **Наслідки:**

1. `VirtualizedList` думає, що кожен елемент змінився.
2. `React.memo` (або `PureComponent`) у `renderItem` не працює.
3. Повний ре-рендер усіх видимих елементів при кожному скролі.

Висновок: Якщо треба трансформувати дані, робіть це заздалегідь або використовуйте мемоізацію.

Концепція "Windowing" (Вікно віртуалізації)

Як це працює: `VirtualizedList` не рендерить весь список. Він підтримує "вікно" рендерингу (Render Window), яке рухається разом зі скролом.

Зони списку:

1. **Leading (Зверху):** Елементи, що вже проскролені. Вони замінюються на порожній простір для звільнення пам'яті.
2. **Visible (Видима):** Те, що бачить користувач прямо зараз (Viewport).
3. **Buffer (Буфер):** Елементи трохи зверху і трохи знизу від вьюпорта, які вже відрендерені, щоб при різкому скролі користувач не побачив білий екран.
4. **Trailing (Знизу):** Майбутні елементи. Вони ще не існують (віртуальні).

windowSize

Що це таке: Prop `windowSize` визначає розмір області, де контент зберігається в пам'яті (відрендереним). Це число вимірюється в **висотах екрана** (screens), а не в пікселях чи елементах.

Математика дефолту: За замовчуванням `windowSize={21}`. Це означає:

- 1 екран видимий.
- 10 екранів зверху (буфер).
- 10 екранів знизу (буфер).

Trade-off (Компроміс):

- **Менше число (напр. 5):** Менше споживання пам'яті RAM, але вищий ризик побачити "білі плями" при швидкому скролі.
- **Більше число (напр. 50):** Скрол ідеально плавний, але список споживає пам'ять і може викликати краш додатку (OOM - Out Of Memory) на слабких Android-девайсах.

"Spacer Views"

Механіка: `VirtualizedList` підтримує внутрішні компоненти-розпірки (Spacers). Коли елементи зверху видаляються, список динамічно збільшує `paddingTop` внутрішнього контейнера на висоту видалених елементів. Коли елементи знизу ще не створені, список тримає `paddingBottom`, щоб емулювати загальну висоту контенту.

Саме для коректного розрахунку цих "розпірок" нам і потрібен `getItemLayout`. Без нього список змушений вгадувати або вимірювати на ходу, що викликає "стрибки".

Керування навантаженням: maxToRenderPerBatch

Проблема: Коли користувач різко скролить вниз, треба терміново відрендерити 50 нових елементів. Якщо React спробує зробити це за один кадр, JS-потік заблокується на 200мс, і анімація скролу зависне.

Рішення: Рендеринг розбивається на пакети (batches). `maxToRenderPerBatch={10}` (дефолт) — означає "рендерити не більше 10 елементів за один такт Event Loop".

Налаштування:

- Для простих елементів (текст) можна збільшити до 20-30.
- Для складних (картинки, тіні, вкладеність) краще зменшити до 5-8, щоб зберегти 60 FPS.

Частота оновлень: updateCellsBatchingPeriod

Призначення: Цей проп задає затримку (в мілісекундах) між рендерингом пакетів. Це своєрідний `throttle` для процесу оновлення списку.

Дефолт: `50` (мс).

Як це впливає:

- **Високе значення (напр. 100+):** Список менше навантажує процесор, інтерфейс чутливіший до дотиків, але користувач частіше бачитиме порожні місця при скролі (контент не встигає з'являтися).
- **Низьке значення (напр. 10):** Контент з'являється миттєво, але JS-потік постійно зайнятий рендерингом, що може призвести до лагів анімацій або пропуску кадрів скролу.

Перший кадр: initialNumToRender

Чому це важливо:

Це єдиний проп, який впливає на **Time-to-Interactive (TTI)**.

VirtualizedList рендерить ці перші N елементів **в тому ж кадрі**, що й сам список (синхронно). Всі наступні елементи рендеряться асинхронно (batching).

Оптимальне значення:

Повинно покривати рівно один екран + 1-2 елементи.

- *Замало:* Користувач бачить порожнє місце знизу при першому завантаженні.
- *Забагато:* JS-потік блокується надовго при переході на екран. Навігація "фризиться".

Формула:

$$Initial = \lceil \frac{ScreenHeight}{ItemHeight} \rceil + 1$$

Трекінг видимості: viewabilityConfig

Сценарій: Автоматичне відтворення відео, коли воно потрапляє в центр екрана, або відправка аналітики "User saw Ad".

API: `onViewableItemsChanged` — колбек, що повертає список видимих ID. `viewabilityConfig` — об'єкт налаштувань.

Конфігурація (itemVisiblePercentThreshold):

- `50`: Елемент вважається видимим, якщо 50% його пікселів на екрані.
- `100`: Елемент має бути повністю на екрані.

Критична помилка: Ніколи не створюйте `viewabilityConfig` всередині рендера!

```
// ❌ BAD: Re-created on every render
viewabilityConfig={{ itemVisiblePercentThreshold: 50 }}

// ✅ GOOD: Defined outside or via useRef
const viewabilityConfig = useRef({ itemVisiblePercentThreshold: 50 }).current;
```

Якщо посилання змінюється, список скидає стан видимості і перераховує все з нуля.

Нативна оптимізація: removeClippedSubviews

Це проп, який спускається до нативного `RCTView`. Якщо `true`, React Native **від'єднує** нативні вьюпорти, які вийшли за межі екрана, від дерева відображення, хоча в JS-пам'яті вони залишаються змонтованими.

Ефект:

- **Android:** Колосальний приріст продуктивності та економія пам'яті. Обов'язково `true` для довгих списків.
- **iOS:** Працює складніше. Може викликати мерехтіння при швидкому скролі, якщо Layout не розрахований ідеально.

Налагодження (Debugging)

Інструмент: Як зрозуміти, що саме віртуалізується, а що ні? У `VirtualizedList` (і `FlatList`) є проп **debug**.

Що він робить: Вмикає візуальний оверлей (overlay) прямо поверх вашого списку в додатку. Він показує:

1. Розміри вікна віртуалізації.
2. Які елементи зараз завантажені в пам'ять.

Для чого використовувати: Для експериментів з `windowSize` або `getItemLayout` коли відбувається дивна поведінка (стрибки, білі екрани), потрібно увімкнути `debug={true}`.