REACT NATIVE MOBILE CUSTOM DAILY

Understanding React Native's FlatList, ScrollView, and SectionList components

Improve Daily video call performance in React Native apps using list components



Jess Mitchell
Jun 13, 2023 / 10 min read



Many developers who build apps with React will encounter React Native at some point in their

careers. React Native allows web developers to build mobile apps – both iOS and Android – with just one codebase written (mostly) in JavaScript.

React Native code will look familiar to developers who are comfortable with React; however, they'll notice many of the component names are a little different.

Since React Native code does not get rendered in the DOM (Document Object Model), it does not render HTML elements, like `<div>`, `<p>`, and so on. Instead, it has components *similar* to HTML elements that will be rendered as native iOS or Android elements. Instead of a `<div>`, React Native uses `View`. Instead of a `<p>` (paragraph), React Native uses `Text` elements that can be styled as needed. Because of this, one of the main learning curves for React developers writing React Native code is learning the appropriate components to use for their mobile apps.

Today's agenda

In this post, we'll focus specifically on React Native list components. A list component is one that has any collection of similar items grouped together. (Think of an unordered or ordered list in HTML – `<ul>` / `<ol>` – for comparison.)
The three React Native components we'll review are:

- `ScrollView`
- `FlatList`
- `SectionList`

This post is not specific to building video or audio apps with Daily. Nevertheless, in the last section we'll review how to improve Daily app performance when you're displaying large lists of videos in a multi-participant call.

TL;DR

Before getting into the details, let's summarize the highlights of what's to come for those of you in a rush:

- A `ScrollView` is a scrolling container, but it's not ideal as a container for mapping over a large collection of list items. This is because it will render the entire list of elements whether they're on-screen or not.
- A `FlatList` component only renders items on screen, which helps improve app performance for long lists.
- A `SectionList` is similar to a `FlatList`, but it is the better option when your list items can be split into subcategories.
- Daily video apps should use a `FlatList` or `SectionList` when rendering large multi-participant calls.

You can improve performance even more in large calls by manually handling media track subscriptions in a React Native Daily call. This means the developer writes logic into their app to subscribe to/unsubscribe from media tracks as the list of viewable call participants changes.

Now, let's get into the specifics.

Types of React Native list components

ScrollView

A `ScrollView` is a generic container component, similar to a `div` with an `overflow` property on it (i.e., it's scrollable when the child elements are too long to fit on-screen). It is *not* specifically a list component but we're including it here because it sometimes gets used like one.

`ScrollView` components – like `div`s – are useful because of how versatile they are. Additionally, since mobile screens are relatively small, it's common to have more content than will fit on the screen, so they're used often.

Let's look at an example of how a `ScrollView` can be used. In the following example, we have some text and a large set of images. If you've never looked at React Native code but are familiar with React, think of the `Text` component as an HTML `<p>` element, and the `Image` component as an HTML `<img>` element. For the sake of simplicity, we'll use the same image source, but pretend it's an app that renders lists of photos, like Instagram.

```
import React from "react";
import { Image, ScrollView, Text } from "react-native";


const src = {
 uri: "https://your-domain.com/Ted.png",
 width: 96,
 height: 96,
};


const textStyles = { fontSize: 56 };
const containerStyles = { padding: 32 };
const imageStyles = { marginBottom: 8 };


const SampleImageList = () =>
 Array(5)
   .fill(src)
   .map((p, i) => <Image style={imageStyles} source={p} key={i} />);


// Let’s pretend these images have different sources
const App = () => (
 <ScrollView style={containerStyles}>
   <Text style={textStyles}>Header 1</Text>
   {SampleImageList()}
   <Text style={textStyles}>Header 2</Text>
   {SampleImageList()}
   <Text style={textStyles}>Header 3</Text>
   {SampleImageList()}
   <Text style={textStyles}>Header 4</Text>
   {SampleImageList()}
 </ScrollView>
);


export default App;
```

This code will look like this in an app:



As we can see, this works! A scrollable list of text and images is rendered as intended. So, what's the problem?

Using a `ScrollView` in this situation gets the job done but it's not the *best* option. It's like using a piece of rope for a belt. Your pants stay up but there's actually an option *designed* for that specific use case.

One of the downsides of this usage is that the entire list renders whether you scroll or not. When it comes to shorter lists, that might not be a big deal, but if you have hundreds or thousands of items being rendered, you'll want to use a more performant option!

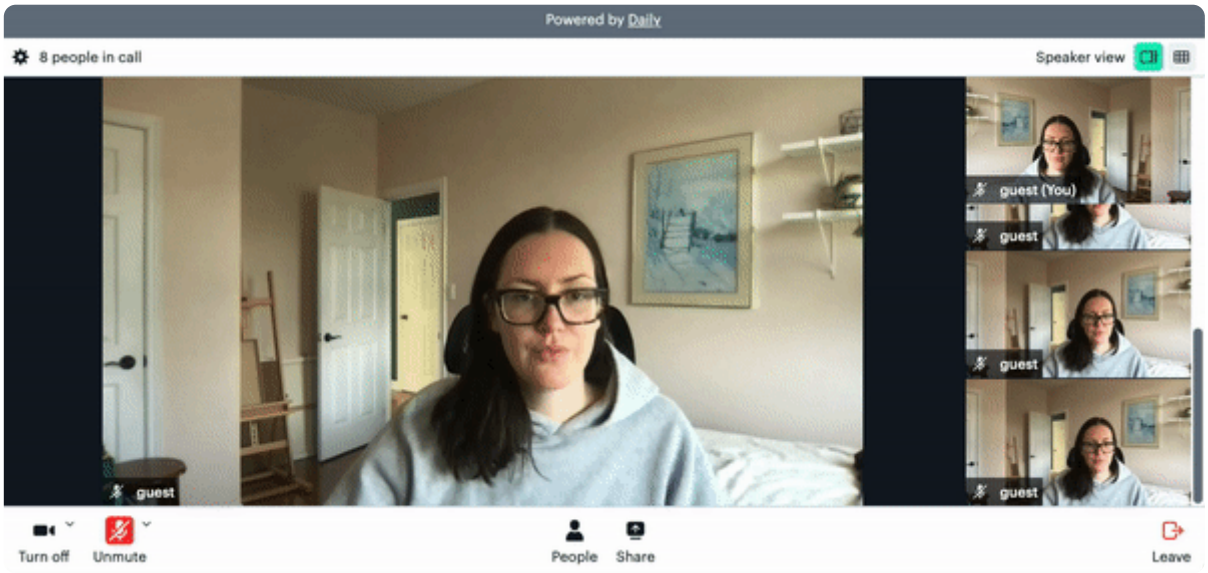
Now, let's look at an actual list component: the `FlatList`.

FlatList

A `FlatList` is a React Native list component that uses virtual scrolling to improve app performance. This means that any list items it renders that are not on the screen will be rendered as blank space until the item is scrolled into the render window.

Note that React Native lets developers determine when an item is considered "viewable" or in the "render window". See the `viewabilityConfig` prop for more information.

Virtual scrolling is an alternative to pagination; it provides a way of taking a list of items that would be too long to render all at once, and allows the items to only be rendered as needed.



A `FlatList` component has two required props:

- A `data` prop, which accepts an array of items to display. (This is the full list – not just the items that are visible.)
- A `renderItem` prop, which lets you create a custom component to render for each item.

The assumption with these props is that the components being rendered will be the same (or at least similar). For example, if you think of a video call app, it's common for each video tile in a multi-participant call to look nearly identical. There is room for variation – like UI changes when someone mutes their audio – but the overall styling is similar. Another example is a list of products in a grocery store app. Some items will have price by weight or quantity, but there's still an image, name, and price for all items.

That being said, since you can render a custom component for each item, you can make the logic for the component as simple or complex as you need it to be.

Now let's look at how `FlatList` code looks using the same page content as before:

```
import React from "react";
import { FlatList, Image, View, Text } from "react-native";


const textStyles = { fontSize: 56 };
const imageStyles = { marginBottom: 8 };


const SampleImageList = () =>
 Array(5)
   .fill(src)
   .map((p, i) => <Image style={imageStyles} source={p} key={i} />);


// Image source
const src = {
 uri: "https://your-domain.com/Ted.png",
 width: 96,
 height: 96,
};
```

```
// Sample data
const imageData = [
 { src: src, title: "Header 1" },
 { src: src, title: "Header 2" },
 { src: src, title: "Header 3" },
 { src: src, title: "Header 4" },
];

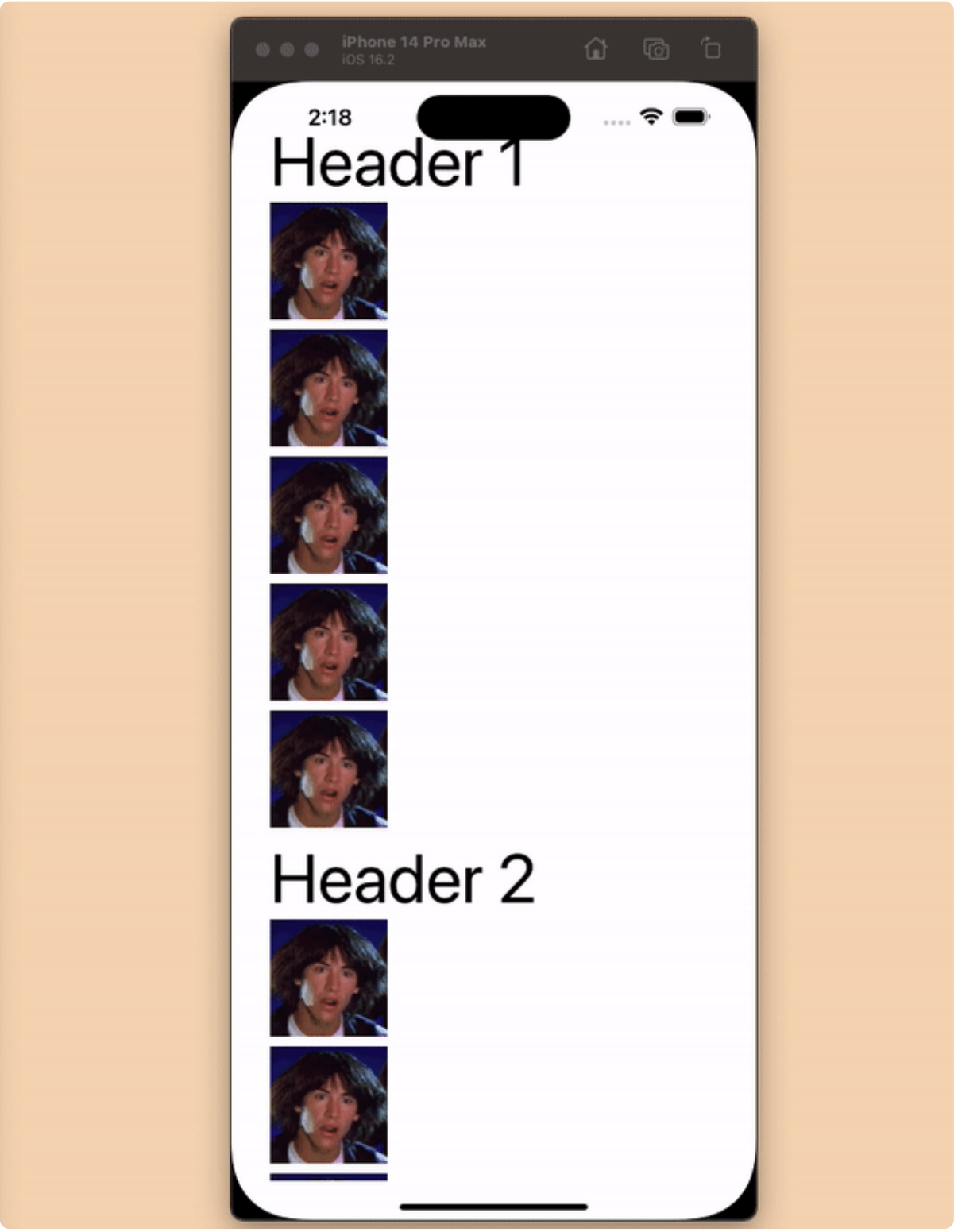

// In your code, you're probably not rendering the same picture 5 times over,
but just go with it
const Item = ({ title, src }) => (
 <View>
   <Text style={textStyles}>{title}</Text>
   {SampleImageList()}
 </View>
);


const App = () => (
 <View style={{ padding: 32 }}>
   <FlatList
     data={imageData}
     renderItem={({ item }) => <Item title={item.title} src={item.src} />}
   />
 </View>
);


export default App;
```

As we can see, the `FlatList` takes two main pieces of information: the `data` (or list of items) and the component that is supposed to be rendered for each item. It also takes a long list of optional props, but we'll stick to the basics here.

Now let's see how this looks:



Unsurprisingly, it looks the same!

Let's review the benefits of the `FlatList` implementation:

- Only items in the render window are rendered, and items are rendered/removed as the user scrolls. This helps with the initial load time and app performance as users interact with it.

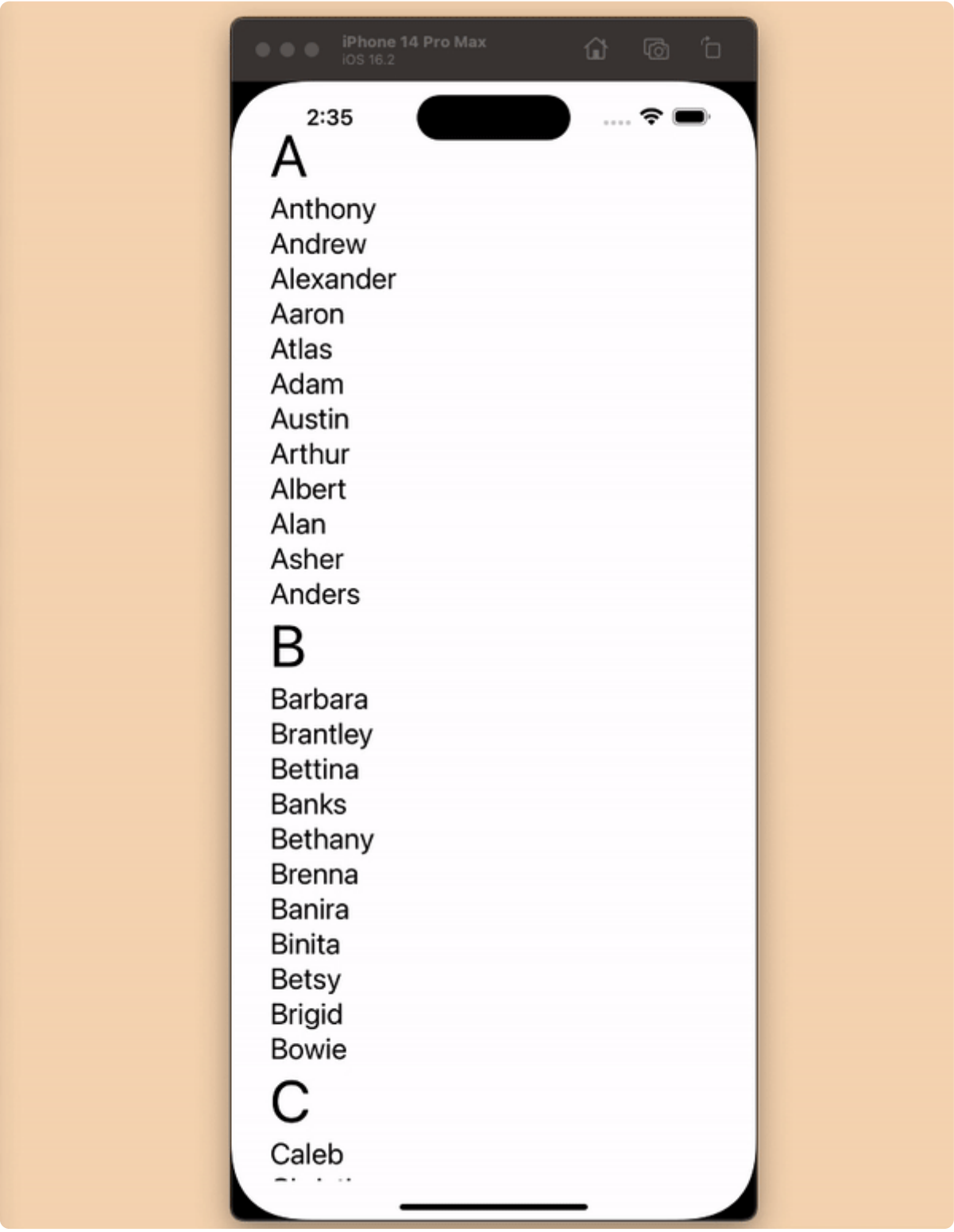
- The `FlatList` component inherits the properties of the `VirtualizedList` component and `ScrollView` component, so you get everything you would with a `ScrollView`, plus more. (Note: The `VirtualizedList` component is the base component for the `FlatList` and `SectionList` components. You *typically* wouldn't use it directly, though.)
- Looking at the code, you immediately know it's a list because the component is a `FlatList`. In other words, there's semantic value to using it. This helps developers newer to the codebase immediately know what they're looking at instead of having to figure it out. (Just like how an HTML `<button>` is quicker to interpret than putting an `onclick` event on a `<div>` ... which developers should generally avoid.)

In short, the `FlatList` is built exactly for this purpose *and* reduces any unnecessary app memory usage on your device.

SectionList

The `SectionList` component is just like a `FlatList` component, but it offers extra features related to list headers and footers, item separators, and section separators. It's intended more for UIs that display a large list in multiple sections.

One simple example of a sectioned list is a contact list in your phone that is separated into alphabetized sections.



In this case, instead of rendering one big list of names, we can improve the user experience of parsing it by adding the letter header to the start of each new section.

In our original example, we also had headers splitting up our list of images, so that tells us our original example is also a good candidate for a `SectionList`. Let's rewrite our `FlatList` code example with a `SectionList` component now.

```
import { Image, SectionList, Text, View } from "react-native";


// Image source
const src = {
 uri: "https://your-domain.com/Ted.png",
 width: 96,
 height: 96,
};


const textStyles = { fontSize: 48 };
const smallTextStyles = { fontSize: 24 };
const containerStyles = { padding: 32 };
const imageStyles = { marginBottom: 8 };


// sample data
const listData = [
 {
   title: "Header 1",
   data: Array(5).fill(src),
 },
 {
   title: "Header 2",
   data: Array(5).fill(src),
 },
 {
   title: "Header 3",
   data: Array(5).fill(src),
 },
 {
   title: "Header 4",
   data: Array(5).fill(src),
 },
];


const Item = ({ item }) => <Image source={item} style={imageStyles} />;
```

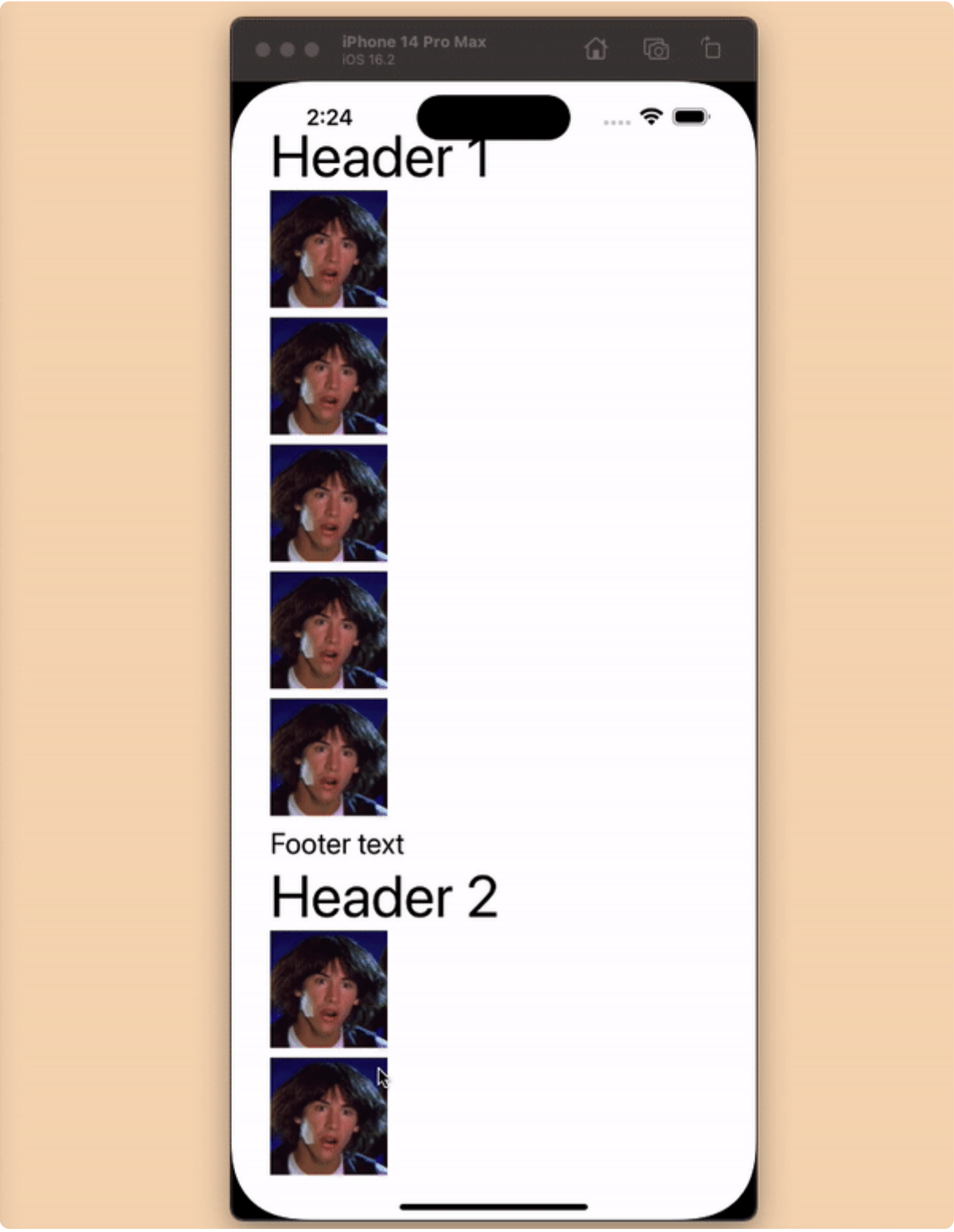
```
const Header = ({ section: { title } }) => (
 <Text style={textStyles}>{title}</Text>
);


const Footer = () => <Text style={smallTextStyles}>Footer text</Text>;


const App = () => (
 <View style={containerStyles}>
   <SectionList
     stickySectionHeadersEnabled={false} // default on ios = true, on android =
false
     sections={listData}
     keyExtractor={(item, index) => item + index}
     renderItem={Item}
     renderSectionHeader={Header}
     renderSectionFooter={Footer}
   />
 </View>
);


export default App;
```

With this new version, we can make the code more semantic by clearly naming our `Item` and `Header` components, which get passed to the `SectionList` component. We even added a `Footer` element to each section to show how nicely this component can handle separating different sections of the list.



Using React Native list components in Daily apps

We've intentionally kept this post more general so it's about lists in React Native instead of specifically using list components in Daily video apps. *That being said,* there are some

interesting options to note if you are building a React Native video app with `react-native-daily-js` – Daily's React Native library.

In video calls that have multiple call participants (and especially ones with an unknown number of possible participants!) we typically recommend using a `FlatList` or `SectionList` component. This allows the video call size to grow while still only rendering videos for the participants who are on-screen. It's important to note that specifically in video call apps, video elements are relatively "expensive" in terms of app memory, so any effort to improve performance will mean a better user experience for your video call participants.

Using track subscriptions

By default, the media tracks for all call participants are subscribed to when using Daily's client SDKs, including `react-native-daily-js` . This is because we try to handle as much of the video call logic as possible on our end to keep things simple for developers. (As video experts, we aim to handle the nitty gritty moving parts of WebRTC instrumentation so you don't have to.)

In addition to using a proper list component, developers building apps for large video calls are encouraged to use Daily's manual track subscription features. Manual track subscription refers to using Daily's SDKs to manually subscribe to relevant video and audio tracks instead of automatically subscribing to all of them.

The main benefit of manual track subscriptions is that developers don't need to waste app memory receiving track updates for participants whose tracks aren't actually being used (e.g., those who are not visible on screen). Calls like webinars and presentations where certain participants are in a "listener" role are other examples of when you might not need to subscribe to everyone's tracks by default.

If you are already using a virtualized list (e.g., a `FlatList` or `SectionList`) and aren't rendering certain videos, you can improve app performance even more by not subscribing to media tracks that are not currently being used.

In terms of how to know when to manually update track subscriptions, the `FlatList` and `SectionList` components both offer a component prop that allows developers to know which items are on screen. The `onViewableItemsChanged` prop accepts a callback function with the `changed` and `viewableItems` arguments. This callback will be invoked whenever the viewable

items change, i.e., whenever a user scrolls and updates the viewable components. This offers a convenient way to know when to check whose tracks can be subscribed to or unsubscribed from. When a participant goes from viewable to not viewable, the developer can unsubscribe, and vice versa when a participant's video becomes visible on scroll.

```
const App = () => (
 <View style={{ padding: 32 }}>
   <FlatList
     data={imageData}
     onViewableItemsChanged={({ viewableItems, changed ] => {
         // Decide how to handle changed and viewableItems 🙂
     }}
     renderItem={({ item }) => <Item title={item.title} src={item.src} />}
   />
 </View>
);
```

Manual track subscriptions is a fairly large topic in itself, so we won't go into more detail here. To learn more, read our blog post on track subscriptions and try our demo app on the topic. The Daily docs for the `updateParticipant()` method also includes additional information.

Conclusion

In today's post, we learned about the `ScrollView`, `FlatList`, and `SectionList` React Native components. We discussed that the latter two options are typically preferable any time a developer is trying to improve app performance when rendering long lists of items/elements.

Specific to Daily video apps, we learned that virtual scrolling is useful to avoid rendered videos that aren't visible. We also reviewed how using manual track subscription can improve app performance further.

To learn more about Daily's client SDK for React Native, read our reference docs and React Native tutorials. To learn more about track subscriptions, read our blog posts on the topic.

And, as always, please reach out to our team if you are building with Daily and could use some help!

Never miss a story

Get the latest direct to your inbox.

Email*

Country*

Join peerConnection - our video/WebRTC developer community.

By submitting, you consent to allow Daily.co to store and process the personal data from this form to fulfill your request.

Subscribe



daily

Interactive Live Streaming for Dummies: Daily.co Special Edition

Get your free copy

Startup program

Use Cases

Telehealth

Live Commerce

EdTech

Event Tech

Creator Technology

Virtual Collaboration

Social and Gaming

Read case studies

Dev Resources

Blog

Guides

Demos

Support

API status

Changelog

Contact us

Contact Us

Live chat

Support

Japan

Resources

Blog

Resource center

Events

Twillio alternative

Vonage alternative

Agora alternative

Videosaurus

Site index

Security & Privacy

Security center

Privacy policy

Terms of service

Cookie policy