Tips

25

25

Pavneet Singh

25

Display a List Using the FlatList Component in React Native

Pavneet Singh

Sep 18, 2020 • 11 Min read • 15,784 Views

Sep 18, 2020 • 11 Min read • 15,784 Views

Web Development Front End Web Dev... Client-side Framew... React

Introduction

Lists are one of the common scrollable components to display similar types of data objects. A list is like an enhanced version of a `ScrollView` component to display data. React Native provides a `FlatList` component to create a list. `FlatList` only renders the list items that can be displayed on the screen. Additionally, `FlatList` offers many inbuilt features like vertical/horizontal scrolling, header/footer views, separator, pull to refresh, lazy

loading, etc. This guide will explain the important details to create and optimize a list of cats images using the TheCatsAPI in React

tive.

Tips

25

25

Basics of FlatList

FlatList is a specialized implementation of the `VirtualizedList` component to display a limited number of items that can fit inside the current window. The rest of the items will be rendered with the list scrolling action. FlatList can simply be implemented using the `data` and `renderItem` props to create a list. There are many other optional props available to add more features, so the props can be categorized into *primary* and *optional*.

Primary Props

- `data` takes an array of items, of type `any`, to populate items in the list.
- `renderItem` requires a function that takes an item object as an input from the `data` source to construct and return a list-item component.

Optional Props

Tips

25

25

Optional props are used to decorate FlatList using an item-divider, header/footer, pull-to-refresh, handle refreshing or optimization ic:

- `ItemSeparatorComponent` is used to add a separator to visually separate items.
- `keyExtractor` is used to provide a unique value (ID, email, etc.) to avoid the recreation of the list by tracking the reordering of the items.
- `extraData` takes a `boolean` value to re-render the current list. FlatList is a `PureComponent` that does not re-render against any change in the `state` or `props` objects, so `extraData` prop is used to re-render the current list if the `data` array is being mutated.
- `initialNumToRender` is used to render a minimum number of item-components in a FlatList for efficiency.
- `ListEmptyComponent` is used to display an empty view while the data is being downloaded.
- `ListHeaderComponent` is used to add a header component like search, menu-items, etc.
- `ListFooterComponent` is used to add a footer component like total items, data summary etc.
- `getItemLayout` returns the predefined size of the list-item component to skip the size calculation process at runtime to speed up the rendering process.
- `horizontal` prop takes a `boolean` value to create a horizontal list by returning like `horizontal={true}`.
- `numColumns` is used to create a column-based list.

- `onRefresh` and `refreshing` are used to implement pull-to-refresh controls, and maintain their visibility using a `boolean` flag.
- `onEndReached` and `onEndReachedThreshold` are used to implement lazy loading callback with a given threshold value. There are other props that can be used to implement style, scrolling, etc.

Downloading List Data

In order to fetch a mock response, use the `search` request API of TheCatsAPI, which doesn't require any registration or API key. The network request can simply be implemented using a `fetch` request:

JSX

```
fetchCats() {
  fetch('https://api.thecatapi.com/v1/images/search?limit=10&page=1') // 1
      .then(res => res.json()) // 2
      .then(resJson => {
          this.setState({ data: resJson }); // 3
      }).catch(e => console.log(e));
}
```

The above request will:

1. Download the data with a `limit` of ten images from `page` one.
2. Convert the response into a JSON object.

3. Store the converted response JSON object in the `state` object as a value of the `data` key by using the `setState` function.

Tips

25

25

lement a List Item Builder Function

The `fetch` request will provide the data to one of the primary props of the `FlatList`. Now implement a function to return a list-item component for the `renderItem` prop:

JSX

```
renderItemComponent = (itemData) => // 1
  <TouchableOpacity>   // 2
      <Image style={styles.image} source={{ uri: itemData.item.url }} /> ,
  </TouchableOpacity>
```

The key field in the response is the `url` that will be used to download and display the image. The following steps explain the working of the `renderItemComponent` function:

1. Takes an object as a parameter that will be used by the UI components.
2. A `TouchableOpacity` component is used to implement a click listener because the `Image` component does not have an `onPress` prop.
3. An `Image` component takes a URI as a source data to display the image. It will download the image automatically.

Tips

25

25

Implement a FlatList

port the `FlatList` component from the `react-native` module
create a list using `data` and `renderItemComponent`:

j

```
render() {
  return (
    <SafeAreaView>
      <FlatList
        data={this.state.data}
        renderItem={item => this.renderItemComponent(item)}
      />
    </SafeAreaView>)
}
```

The `this.state.data` will be updated by the `fetch` request, and it will trigger a re-rendering process in the `FlatList` component. The above implementation works fine, but the next steps will add more features to improve the `FlatList` implementation.

Implement KeyExtractor

React always uses a unique key to track the updates in a component. By default, the `FlatList` looks either for a custom `keyExtractor` implementation or a field named `key` in a data item, otherwise it uses the array index as the value of `key`. The `id` is a

Tips

25

unique value in the response that can be used to implement the `keyExtractor`:

jsx

```
render() {
  return (
    <SafeAreaView>
      <FlatList
        data={this.state.data}
        renderItem={item => this.renderItemComponent(item)}
        keyExtractor={item => item.id.toString()}
      />
    </SafeAreaView>)
}
```

Implement a Separator

A separator is a component that is placed between the list-items. It can be implemented by using a `View` component:

JSX

```
ItemSeprator = () => <View style={{
  height: 2,
  width: "100%",
  backgroundColor: "rgba(0,0,0,0.5)",
}} />

<FlatList
    //...
```

```
        ItemSeparatorComponent={this.ItemSeprator}
    />
```

Tips

25

> *Just like `ItemSeparatorComponent`, different components can be implemented for `ListFooterComponent` and `ListHeaderComponent` props.*

Implement Pull-to-refresh

The pull-to-refresh implementation requires a `boolean` flag for the `refreshing` prop to hide/display the loading indication. The `handleRefresh` is a function for the `onRefresh` prop to update the data and loading indicator:

JSX

```
handleRefresh = () => {
    this.setState({ refreshing: false }, ()=>{this.fetchCats()});
}
<FlatList
    //...
    refreshing={this.state.refreshing}
    onRefresh={this.handleRefresh}
/>)
```

The value of `refreshing` is set to `true` before the `fetch` call to display the loading indicator. The value of the `refreshing` field will be set to `false` in case of a successful response or error.

Handle Mutable Data Changes

Since `FlatList` is a `PureComponent`, it assumes that the data is an immutable object and that any change in data is done by setting a new data source object in the state object. Alternatively, use the `extraData={this.state.isChanged}` prop to re-render a `FlatList` component by updating the value of `isChanged` property.

Here's the complete code of the `App.js` component:

JSX

```
/**
 * @author Pavneet Singh
 */

import React from "react";
import {
    StyleSheet,
    SafeAreaView,
    FlatList,
    View,
    Image,
    TouchableOpacity
} from "react-native";

export default class App extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            data: [],
            refreshing: true,
```

Tips

25


```
        }
    }

    componentDidMount() {
        this.fetchCats();
    }

    fetchCats() {
        this.setState({ refreshing: true });
        fetch('https://api.thecatapi.com/v1/images/search?limit=10&page=1
            .then(res => res.json())
            .then(resJson => {
                this.setState({ data: resJson });
                this.setState({ refreshing: false });
            }).catch(e => console.log(e));
    }

    renderItemComponent = (data) =>
        <TouchableOpacity style={styles.container}>
            <Image style={styles.image} source={{ uri: data.item.url }} /
        </TouchableOpacity>

    ItemSeparator = () => <View style={{
        height: 2,
        backgroundColor: "rgba(0,0,0,0.5)",
        marginLeft: 10,
        marginRight: 10,
    }}
    />

    handleRefresh = () => {
        this.setState({ refreshing: false }, () => { this.fetchCats() });
    }

    render() {
```

Tips

25


```
56          return (
57            <SafeAreaView>
58              <FlatList
59                data={this.state.data}
60                renderItem={item => this.renderItemComponent(item)}
61                keyExtractor={item => item.id.toString()}
62                ItemSeparatorComponent={this.ItemSeparator}
63                refreshing={this.state.refreshing}
64                onRefresh={this.handleRefresh}
65              />
66            </SafeAreaView>)
67        }
68      }
69
70      const styles = StyleSheet.create({
71        container: {
72          height: 300,
73          margin: 10,
74          backgroundColor: '#FFF',
75          borderRadius: 6,
76        },
77        image: {
78          height: '100%',
79          borderRadius: 4,
80        },
81      });
```

Tips

Tips

25

25

- Use the `initialNumToRender` prop to render the defined numbers of items initially, and `maxToRenderPerBatch` to render items for every scroll to avoid blank space on the screen during scrolling. The default value for both of the props is `10`.
- Use the `getItemLayout` prop if the height or width of list-item is fixed. For example, if the height for item is 100 then use:

JSX

```
getItemLayout={(data, index) => (
  {length: 100, offset: 100 * index, index}
)}
```

The `length` used to define item height (vertically) and `offset` is used to define the starting point for the current item index. For example, the starting point of the third item will be `100 * 2 = 200` (array index starts from 0).

Conclusion

`FlatList` offers great flexibility to create and configure a list in a React Native app. `FlatList` can be optimized for static and dynamic data using various layout calculations and rendering props. The React Native team is working to further improve its

performance for large datasets. The optimized project is available on my RNList repository. Happy coding!

25

Tips

25