Version: 6.x

React Native Tab View

React Native Tab View is a cross-platform Tab View component for React Native implemented using `react-native-pager-view` on Android & iOS, and PanResponder on Web, macOS, and Windows.

It follows material design guidelines by default, but you can also use your own custom tab bar or position the tab bar at the bottom.

This package doesn't integrate with React Navigation. If you want to integrate the tab view with React Navigation's navigation system, e.g. want to show screens in the tab bar and be able to navigate between them using `navigation.navigate` etc, use Material Top Tab Navigator instead.

Installation

To use this package, open a Terminal in the project root and run:

npm Yarn

```
npm install react-native-tab-view
```

Next, install `react-native-pager-view` if you plan to support iOS and Android.

If you are using Expo, to ensure that you get the compatible versions of the libraries, run:

```
expo install react-native-pager-view
```

If you are not using Expo, run the following:

npm Yarn

```
npm install react-native-pager-view
```

We're done! Now you can build and run the app on your device/simulator.

Quick start

```
import * as React from 'react';
import { View, useWindowDimensions } from 'react-native';
import { TabView, SceneMap } from 'react-native-tab-view';

const FirstRoute = () => (
  <View style={{ flex: 1, backgroundColor: '#ff4081' }} />
);

const SecondRoute = () => (
  <View style={{ flex: 1, backgroundColor: '#673ab7' }} />
);

const renderScene = SceneMap({
  first: FirstRoute,
  second: SecondRoute,
});

export default function TabViewExample() {
  const layout = useWindowDimensions();

  const [index, setIndex] = React.useState(0);
  const [routes] = React.useState([
    { key: 'first', title: 'First' },
    { key: 'second', title: 'Second' },
  ]);

  return (
    <TabView
      navigationState={{ index, routes }}
      renderScene={renderScene}
      onIndexChange={setIndex}
      initialLayout={{ width: layout.width }}
    />
  );
}
```

Try this example on Snack

More examples on Snack

- Custom Tab Bar
- Lazy Load

API reference

The package exports a `TabView` component which is the one you'd use to render the tab view, and a `TabBar` component which is the default tab bar implementation.

`TabView`

Container component responsible for rendering and managing tabs. Follows material design styles by default.

Basic usage look like this:

```
<TabView
  navigationState={{ index, routes }}
  onIndexChange={setIndex}
  renderScene={SceneMap({
    first: FirstRoute,
    second: SecondRoute,
  })}
/>
```

TabView Props

`navigationState` (`required`)

State for the tab view. The state should contain the following properties:

- `index`: a number representing the index of the active route in the `routes` array
- `routes`: an array containing a list of route objects used for rendering the tabs

Each route object should contain the following properties:

- `key`: a unique key to identify the route (required)
- `title`: title for the route to display in the tab bar
- `icon`: icon for the route to display in the tab bar
- `accessibilityLabel`: accessibility label for the tab button
- `testID`: test id for the tab button

Example:

```
{
  index: 1,
  routes: [
    { key: 'music', title: 'Music' },
    { key: 'albums', title: 'Albums' },
    { key: 'recents', title: 'Recents' },
    { key: 'purchased', title: 'Purchased' },
  ]
}
```

`TabView` is a controlled component, which means the `index` needs to be updated via the `onIndexChange` callback.

`onIndexChange` (`required`)

Callback which is called on tab change, receives the index of the new tab as argument. The navigation state needs to be updated when it's called, otherwise the change is dropped.

`renderScene` (`required`)

Callback which returns a react element to render as the page for the tab. Receives an object containing the route as the argument:

```
const renderScene = ({ route, jumpTo }) => {
  switch (route.key) {
    case 'music':
      return <MusicRoute jumpTo={jumpTo} />;
    case 'albums':
      return <AlbumsRoute jumpTo={jumpTo} />;
  }
};
```

You need to make sure that your individual routes implement a `shouldComponentUpdate` to improve the performance. To make it easier to specify the components, you can use the `SceneMap` helper.

`SceneMap` takes an object with the mapping of `route.key` to React components and returns a function to use with `renderScene` prop.

```
import { SceneMap } from 'react-native-tab-view';

...

const renderScene = SceneMap({
  music: MusicRoute,
  albums: AlbumsRoute,
});
```

Specifying the components this way is easier and takes care of implementing a `shouldComponentUpdate` method.

Each scene receives the following props:

- `route`: the current route rendered by the component
- `jumpTo`: method to jump to other tabs, takes a `route.key` as it's argument
- `position`: animated node which represents the current position

The `jumpTo` method can be used to navigate to other tabs programmatically:

```
props.jumpTo('albums');
```

All the scenes rendered with `SceneMap` are optimized using `React.PureComponent` and don't re-render when parent's props or states change. If you need more control over how your scenes update (e.g. - triggering a re-render even if the `navigationState` didn't change), use `renderScene` directly instead of using `SceneMap`.

IMPORTANT: Do not pass inline functions to `SceneMap`, for example, don't do the following:

```
SceneMap({
  first: () => <FirstRoute foo={props.foo} />,
  second: SecondRoute,
});
```

Always define your components elsewhere in the top level of the file. If you pass inline functions, it'll re-create the component every render, which will cause the entire route to unmount and remount every change. It's very bad for performance and will also cause any local state to be lost.

If you need to pass additional props, use a custom `renderScene` function:

```
const renderScene = ({ route }) => {
  switch (route.key) {
    case 'first':
      return <FirstRoute foo={this.props.foo} />;
    case 'second':
      return <SecondRoute />;
    default:
      return null;
  }
};
```

`renderTabBar`

Callback which returns a custom React Element to use as the tab bar:

```
import { TabBar } from 'react-native-tab-view';

...

<TabView
  renderTabBar={props => <TabBar {...props} />}
  ...
/>
```

If this is not specified, the default tab bar is rendered. You pass this props to customize the default tab bar, provide your own tab bar, or disable the tab bar completely.

```
<TabView
  renderTabBar={() => null}
  ...
/>
```

`tabBarPosition`

Position of the tab bar in the tab view. Possible values are `'top'` and `'bottom'`. Defaults to `'top'`.

`lazy`

Function which takes an object with the current route and returns a boolean to indicate whether to lazily render the scenes.

By default all scenes are rendered to provide a smoother swipe experience. But you might want to defer the rendering of unfocused scenes until the user sees them. To enable lazy rendering for a particular scene, return `true` from `getLazy` for that `route`:

```
<TabView
  lazy={({ route }) => route.name === 'Albums'}
  ...
/>
```

When you enable lazy rendering for a screen, it will usually take some time to render when it comes into focus. You can use the `renderLazyPlaceholder` prop to customize what the user sees during this short period.

You can also pass a boolean to enable lazy for all of the scenes:

```
<TabView
  lazy
/>
```

`lazyPreloadDistance`

When `lazy` is enabled, you can specify how many adjacent routes should be preloaded with this prop. This value defaults to `0` which means lazy pages are loaded as they come into the viewport.

`renderLazyPlaceholder`

Callback which returns a custom React Element to render for routes that haven't been rendered yet. Receives an object containing the route as the argument. The `lazy` prop also needs to be enabled.

This view is usually only shown for a split second. Keep it lightweight.

By default, this renders `null`.

`keyboardDismissMode`

String indicating whether the keyboard gets dismissed in response to a drag gesture. Possible values are:

- `'auto'` (default): the keyboard is dismissed when the index changes.
- `'on-drag'`: the keyboard is dismissed when a drag begins.
- `'none'`: drags do not dismiss the keyboard.

`swipeEnabled`

Boolean indicating whether to enable swipe gestures. Swipe gestures are enabled by default. Passing `false` will disable swipe gestures, but the user can still switch tabs by pressing the tab bar.

`animationEnabled`

Enables animation when changing tab. By default it's true.

`onSwipeStart`

Callback which is called when the swipe gesture starts, i.e. the user touches the screen and moves it.

`onSwipeEnd`

Callback which is called when the swipe gesture ends, i.e. the user lifts their finger from the screen after the swipe gesture.

`initialLayout`

Object containing the initial height and width of the screens. Passing this will improve the initial rendering performance. For most apps, this is a good default:

```
<TabView
  initialLayout={{ width: Dimensions.get('window').width }}
  ...
/>
```

`overScrollMode`

Used to override default value of pager's overScroll mode. Can be `auto`, `always` or `never` (Android only).

`sceneContainerStyle`

Style to apply to the view wrapping each screen. You can pass this to override some default styles such as overflow clipping:

`pagerStyle`

Style to apply to the pager view wrapping all the scenes.

`style`

Style to apply to the tab view container.

`TabBar`

Material design themed tab bar. To customize the tab bar, you'd need to use the `renderTabBar` prop of `TabView` to render the `TabBar` and pass additional props.

For example, to customize the indicator color and the tab bar background color, you can pass `indicatorStyle` and `style` props to the `TabBar` respectively:

```
const renderTabBar = props => (
  <TabBar
    {...props}
    indicatorStyle={{ backgroundColor: 'white' }}
    style={{ backgroundColor: 'pink' }}
  />
);

//...


return (
  <TabView
    renderTabBar={renderTabBar}
    ...
  />
);
```

TabBar Props

`getLabelText`

Function which takes an object with the current route and returns the label text for the tab. Uses `route.title` by default.

```
<TabBar
  getLabelText={({ route }) => route.title}
  ...
/>
```

`getAccessible`

Function which takes an object with the current route and returns a boolean to indicate whether to mark a tab as `accessible`. Defaults to `true`.

`getAccessibilityLabel`

Function which takes an object with the current route and returns a accessibility label for the tab button. Uses `route.accessibilityLabel` by default if specified, otherwise uses the route title.

```
<TabBar
  getAccessibilityLabel={({ route }) => route.accessibilityLabel}
  ...
/>
```

`getTestID`

Function which takes an object with the current route and returns a test id for the tab button to locate this tab button in tests. Uses `route.testID` by default.

```
<TabBar
  getTestID={({ route }) => route.testID}
  ...
/>
```

`renderIcon`

Function which takes an object with the current route, focused status and color and returns a custom React Element to be used as a icon.

```
<TabBar
  renderIcon={({ route, focused, color }) => (
    <Icon
      name={focused ? 'albums' : 'albums-outlined'}
      color={color}
    />
  )}
  ...
/>
```

`renderLabel`

Function which takes an object with the current route, focused status and color and returns a custom React Element to be used as a label.

```
<TabBar
  renderLabel={({ route, focused, color }) => (
    <Text style={{ color, margin: 8 }}>
      {route.title}
    </Text>
  )}
  ...
/>
```

`renderTabBarItem`

Function which takes a `TabBarItemProps` object and returns a custom React Element to be used as a tab button.

`renderIndicator`

Function which takes an object with the current route and returns a custom React Element to be used as a tab indicator.

`renderBadge`

Function which takes an object with the current route and returns a custom React Element to be used as a badge.

`onTabPress`

Function to execute on tab press. It receives the scene for the pressed tab, useful for things like scroll to top.

By default, tab press also switches the tab. To prevent this behavior, you can call `preventDefault`:

```
<TabBar
  onTabPress={({ route, preventDefault }) => {
    if (route.key === 'home') {
      preventDefault();

      // Do something else
    }
  }}
  ...
/>
```

`onTabLongPress`

Function to execute on tab long press, use for things like showing a menu with more options

`activeColor`

Custom color for icon and label in the active tab.

`inactiveColor`

Custom color for icon and label in the inactive tab.

`pressColor`

Color for material ripple (Android >= 5.0 only).

`pressOpacity`

Opacity for pressed tab (iOS and Android < 5.0 only).

`scrollEnabled`

Boolean indicating whether to make the tab bar scrollable.

If you set `scrollEnabled` to `true`, you should also specify a `width` in `tabStyle` to improve the initial render.

`bounces`

Boolean indicating whether the tab bar bounces when scrolling.

`tabStyle`

Style to apply to the individual tab items in the tab bar.

By default, all tab items take up the same pre-calculated width based on the width of the container. If you want them to take their original width, you can specify `width: 'auto'` in `tabStyle`.

`indicatorStyle`

Style to apply to the active indicator.

`indicatorContainerStyle`

Style to apply to the container view for the indicator.

`labelStyle`

Style to apply to the tab item label.

`contentContainerStyle`

Style to apply to the inner container for tabs.

`style` (`TabBar`)

Style to apply to the tab bar container.

`gap`

Define a spacing between tabs.

`testID`

Test id for the tabBar. Can be used for scrolling the tab bar in tests

Optimization Tips

Avoid unnecessary re-renders

The `renderScene` function is called every time the index changes. If your `renderScene` function is expensive, it's good idea move each route to a separate component if they don't depend on the index, and use `shouldComponentUpdate` or `React.memo` in your route components to prevent unnecessary re-renders.

For example, instead of:

```
const renderScene = ({ route }) => {
  switch (route.key) {
    case 'home':
      return (
        <View style={styles.page}>
          <Avatar />
          <NewsFeed />
        </View>
      );
    default:
      return null;
  }
};
```

Do the following:

```
const renderScene = ({ route }) => {
  switch (route.key) {
    case 'home':
      return <HomeComponent />;
    default:
      return null;
  }
};
```

Where `<HomeComponent />` is a `PureComponent` if you're using class components:

```
export default class HomeComponent extends React.PureComponent {
  render() {
    return (
      <View style={styles.page}>
        <Avatar />
        <NewsFeed />
      </View>
    );
  }
}
```

Or, wrapped in `React.memo` if you're using function components:

```
function HomeComponent() {
  return (
    <View style={styles.page}>
      <Avatar />
      <NewsFeed />
    </View>
  );
}

export default React.memo(HomeComponent);
```

Avoid one frame delay

We need to measure the width of the container and hence need to wait before rendering some elements on the screen. If you know the initial width upfront, you can pass it in and we won't need to wait for measuring it. Most of the time, it's just the window width.

For example, pass the following `initialLayout` to `TabView`:

```
const initialLayout = {
  height: 0,
  width: Dimensions.get('window').width,
};
```

The tab view will still react to changes in the dimension and adjust accordingly to accommodate things like orientation change.

Optimize large number of routes

If you've a large number of routes, especially images, it can slow the animation down a lot. You can instead render a limited number of routes.

For example, do the following to render only 2 routes on each side:

```
const renderScene = ({ route }) => {
  if (Math.abs(index - routes.indexOf(route)) > 2) {
    return <View />;
  }

  return <MySceneComponent route={route} />;
};
```

Avoid rendering TabView inside ScrollView

Nesting the `TabView` inside a vertical `ScrollView` will disable the optimizations in the `FlatList` components rendered inside the `TabView`. So avoid doing it if possible.

Use `lazy` and `renderLazyPlaceholder` props to render routes as needed

The `lazy` option is disabled by default to provide a smoother tab switching experience, but you can enable it and provide a placeholder component for a better lazy loading experience. Enabling `lazy` can improve initial load performance by rendering routes only when they come into view. Refer the prop reference for more details.

Edit this page