Version: 6.x

NavigationContainer

The `NavigationContainer` is responsible for managing your app state and linking your top-level navigator to the app environment.

The container takes care of platform specific integration and provides various useful functionality:

1. Deep link integration with the `linking` prop.
2. Notify state changes for screen tracking, state persistence etc.
3. Handle system back button on Android by using the `BackHandler` API from React Native.

Usage:

```
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack';

const Stack = createNativeStackNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>{/* ... */}</Stack.Navigator>
    </NavigationContainer>
  );
}
```

Ref

It's also possible to attach a `ref` to the container to get access to various helper methods, for example, dispatch navigation actions. This should be used in rare cases when you don't have access to the `navigation` prop, such as a Redux middleware.

Example:

```
import { NavigationContainer, useNavigationContainerRef } from '@react-
navigation/native';

function App() {
  const navigationRef = useNavigationContainerRef(); // You can also use a
regular ref with `React.useRef()`

  return (
    <View style={{ flex: 1 }}>
      <Button onPress={() => navigationRef.navigate('Home')}>
        Go home
      </Button>
      <NavigationContainer ref={navigationRef}>{/* ... */}</NavigationContainer>
    </View>
  );
}
```

Try this example on Snack

If you're using a regular ref object, keep in mind that the ref may be initially `null` in some situations (such as when linking is enabled). To make sure that the ref is initialized, you can use the `onReady` callback to get notified when the navigation container finishes mounting.

See the Navigating without the navigation prop guide for more details.

Methods on the ref

The ref object includes all of the common navigation methods such as `navigate`, `goBack` etc. See docs for `CommonActions` for more details.

Example:

```
navigationRef.navigate(name, params);
```

All of these methods will act as if they were called inside the currently focused screen. It's important note that there must be a navigator rendered to handle these actions.

In addition to these methods, the ref object also includes the following special methods:

`resetRoot`

The `resetRoot` method lets you reset the state of the navigation tree to the specified state object:

```
navigationRef.resetRoot({
  index: 0,
  routes: [{ name: 'Profile' }],
});
```

Unlike the `reset` method, this acts on the root navigator instead of navigator of the currently focused screen.

`getRootState`

The `getRootState` method returns a navigation state object containing the navigation states for all navigators in the navigation tree:

```
const state = navigationRef.getRootState();
```

Note that the returned `state` object will be `undefined` if there are no navigators currently rendered.

`getCurrentRoute`

The `getCurrentRoute` method returns the route object for the currently focused screen in the whole navigation tree:

```
const route = navigationRef.getCurrentRoute();
```

Note that the returned `route` object will be `undefined` if there are no navigators currently rendered.

`getCurrentOptions`

The `getCurrentOptions` method returns the options for the currently focused screen in the whole navigation tree:

```
const options = navigationRef.getCurrentOptions();
```

Note that the returned `options` object will be `undefined` if there are no navigators currently rendered.

`addListener`

The `addListener` method lets you listen to the following events:

`state`

The event is triggered whenever the navigation state changes in any navigator in the navigation tree:

```
const unsubscribe = navigationRef.addListener('state', (e) => {
  // You can get the raw navigation state (partial state object of the root
navigator)
  console.log(e.data.state);

  // Or get the full state object with `getRootState()`
  console.log(navigationRef.getRootState());
});
```

This is analogous to the `onStateChange` method. The only difference is that the `e.data.state` object might contain partial state object unlike the `state` argument in `onStateChange` which will always contain the full state object.

`options`

The event is triggered whenever the options change for the currently focused screen in the navigation tree:

```
const unsubscribe = navigationRef.addListener('options', (e) => {
  // You can get the new options for the currently focused screen
  console.log(e.data.options);
});
```

Props

`initialState`

Prop that accepts initial state for the navigator. This can be useful for cases such as deep linking, state persistence etc.

Example:

```
<NavigationContainer
  initialState={initialState}
>
  {/* ... */}
</NavigationContainer>
```

Providing a custom initial state object will override the initial state object obtained via linking configuration or from browser's URL. If you're providing an initial state object, make sure that you don't pass it on web and that there's no deep link to handle.

Example:

```
const initialUrl = await Linking.getInitialURL();

if (Platform.OS !== 'web' && initialUrl == null) {
  // Only restore state if there's no deep link and we're not on web
}
```

See state persistence guide for more details on how to persist and restore state.

`onStateChange`

> Note: Consider the navigator's state object to be internal and subject to change in a minor release. Avoid using properties from the navigation state object except `index` and `routes`, unless you really need it. If there is some functionality you cannot achieve without relying on the structure of the state object, please open an issue.

Function that gets called every time navigation state changes. It receives the new navigation state as the argument.

You can use it to track the focused screen, persist the navigation state etc.

Example:

```
<NavigationContainer
  onStateChange={(state) => console.log('New state is', state)}
>
  {/* ... */}
</NavigationContainer>
```

`onReady`

Function which is called after the navigation container and all its children finish mounting for the first time. You can use it for:

- Making sure that the `ref` is usable. See docs regarding initialization of the ref for more details.
- Hiding your native splash screen

Example:

```
<NavigationContainer
  onReady={() => console.log('Navigation container is ready')}
>
  {/* ... */}
</NavigationContainer>
```

`onUnhandledAction`

Function which is called when a navigation action is not handled by any of the navigators.

By default, React Navigation will show a development-only error message when an action was not handled. You can override the default behavior by providing a custom function.

`linking`

Configuration for linking integration used for deep linking, URL support in browsers etc.

Example:

```
import { NavigationContainer } from '@react-navigation/native';

function App() {
  const linking = {
    prefixes: ['https://mychat.com', 'mychat://'],
    config: {
      screens: {
        Home: 'feed/:sort',
      },
    },
  };

  return (
    <NavigationContainer linking={linking} fallback={<Text>Loading...</Text>}>
      {/* content */}
    </NavigationContainer>
  );
}
```

See configuring links guide for more details on how to configure deep links and URL integration.

Options

`linking.prefixes`

URL prefixes to handle. You can provide multiple prefixes to support custom schemes as well as universal links.

Only URLs matching these prefixes will be handled. The prefix will be stripped from the URL before parsing.

Example:

```
<NavigationContainer
  linking={{
    prefixes: ['https://mychat.com', 'mychat://'],
    config: {
      screens: {
        Chat: 'feed/:sort',
      },
    },
  }}
>
  {/* content */}
</NavigationContainer>
```

This is only supported on iOS and Android.

`linking.config`

Config to fine-tune how to parse the path. The config object should represent the structure of the navigators in the app.

For example, if we have `Catalog` screen inside `Home` screen and want it to handle the `item/:id` pattern:

```
{
  screens: {
    Home: {
      screens: {
        Catalog: {
          path: 'item/:id',
          parse: {
            id: Number,
          },
        },
      },
    },
  }
}
```

The options for parsing can be an object or a string:

```
{
  screens: {
    Catalog: 'item/:id',
  }
}
```

When a string is specified, it's equivalent to providing the `path` option.

The `path` option is a pattern to match against the path. Any segments starting with `:` are recognized as a param with the same name. For example `item/42` will be parsed to `{ name: 'item', params: { id: '42' } }`.

The `initialRouteName` option ensures that the route name passed there will be present in the state for the navigator, e.g. for config:

```
{
  screens: {
    Home: {
      initialRouteName: 'Feed',
      screens: {
        Catalog: {
          path: 'item/:id',
          parse: {
            id: Number,
          },
        },
        Feed: 'feed',
      },
    },
  }
}
```

and URL : `/item/42`, the state will look like this:

```
{
  routes: [
    {
      name: 'Home',
      state: {
        index: 1,
        routes: [
          {
            name: 'Feed'
          },
          {
            name: 'Catalog',
            params: { id: 42 },
          },
        ],
        },
      },
    ],
  }
```

The `parse` option controls how the params are parsed. Here, you can provide the name of the param to parse as a key, and a function which takes the string value for the param and returns a parsed value:

```
{
  screens: {
    Catalog: {
      path: 'item/:id',
      parse: {
        id: id => parseInt(id, 10),
      },
    },
  }
}
```

If no custom function is provided for parsing a param, it'll be parsed as a string.

`linking.enabled`

Optional boolean to enable or disable the linking integration. Defaults to `true` if the `linking` prop is specified.

`linking.getInitialURL`

By default, linking integrates with React Native's `Linking` API and uses `Linking.getInitialURL()` to provide built-in support for deep linking. However, you might also want to handle links from other sources, such as Branch, or push notifications using Firebase etc.

You can provide a custom `getInitialURL` function where you can return the link which we should use as the initial URL. The `getInitialURL` function should return a `string` if there's a URL to handle, otherwise `undefined`.

For example, you could do something like following to handle both deep linking and Firebase notifications:

```
import messaging from '@react-native-firebase/messaging';

<NavigationContainer
  linking={{
    prefixes: ['https://mychat.com', 'mychat://'],
    config: {
      screens: {
        Chat: 'feed/:sort',
      },
    },
    async getInitialURL() {
      // Check if app was opened from a deep link
      const url = await Linking.getInitialURL();

      if (url != null) {
        return url;
      }

      // Check if there is an initial firebase notification
      const message = await messaging().getInitialNotification();

      // Get the `url` property from the notification which corresponds to a
screen
      // This property needs to be set on the notification payload when sending
it
      return message?.data?.url;
    },
  }}
>
  {/* content */}
</NavigationContainer>
```

This option is not available on Web.

`linking.subscribe`

Similar to `getInitialURL`, you can provide a custom `subscribe` function to handle any incoming links instead of the default deep link handling. The `subscribe` function will receive a listener as the argument and you can call it with a URL string whenever there's a new URL to handle. It should return a cleanup function where you can unsubscribe from any event listeners that you have setup.

For example, you could do something like following to handle both deep linking and Firebase notifications:

```
import messaging from '@react-native-firebase/messaging';

<NavigationContainer
  linking={{
    prefixes: ['https://mychat.com', 'mychat://'],
    config: {
      screens: {
        Chat: 'feed/:sort',
      },
    },
    subscribe(listener) {
      const onReceiveURL = ({ url }: { url: string }) => listener(url);

      // Listen to incoming links from deep linking
      const subscription = Linking.addEventListener('url', onReceiveURL);

      // Listen to firebase push notifications
      const unsubscribeNotification = messaging().onNotificationOpenedApp(
        (message) => {
          const url = message.data?.url;

          if (url) {
            // Any custom logic to check whether the URL needs to be handled
            //...

            // Call the listener to let React Navigation handle the URL
            listener(url);
          }
        }
      );

      return () => {
        // Clean up the event listeners
        subscription.remove();
        unsubscribeNotification();
      };
    },
  }}
>
```

```
  {/* content */}
</NavigationContainer>
```

This option is not available on Web.

`linking.getStateFromPath`

You can optionally override the way React Navigation parses links to a state object by providing your own implementation.

Example:

```
<NavigationContainer
  linking={{
    prefixes: ['https://mychat.com', 'mychat://'],
    config: {
      screens: {
        Chat: 'feed/:sort',
      },
    },
    getStateFromPath(path, config) {
      // Return a state object here
      // You can also reuse the default logic by importing `getStateFromPath`
from `@react-navigation/native`
    },
  }}
>
  {/* content */}
</NavigationContainer>
```

`linking.getPathFromState`

You can optionally override the way React Navigation serializes state objects to link by providing your own implementation. This is necessary for proper web support if you have specified `getStateFromPath`.

Example:

```
<NavigationContainer
  linking={{
    prefixes: ['https://mychat.com', 'mychat://'],
    config: {
      screens: {
        Chat: 'feed/:sort',
      },
    },
    getPathFromState(state, config) {
      // Return a path string here
      // You can also reuse the default logic by importing `getPathFromState`
from `@react-navigation/native`
    },
  }}
>
  {/* content */}
</NavigationContainer>
```

`fallback`

React Element to use as a fallback while we resolve deep links. Defaults to `null`.

If you have a native splash screen, please use `onReady` instead of `fallback` prop.

`documentTitle`

By default, React Navigation automatically updates the document title on Web to match the `title` option of the focused screen. You can disable it or customize it using this prop. It accepts a configuration object with the following options:

`documentTitle.enabled`

Whether document title handling should be enabled. Defaults to `true`.

`documentTitle.formatter`

Custom formatter to use if you want to customize the title text. Defaults to:

```
(options, route) => options?.title ?? route?.name;
```

Example:

```
import { NavigationContainer } from '@react-navigation/native';

function App() {
  return (
    <NavigationContainer
      documentTitle={{
        formatter: (options, route) =>
          `${options?.title ?? route?.name} - My Cool App`,
      }}
    >
      {/* content */}
    </NavigationContainer>
  );
}
```

`theme`

Custom theme to use for the navigation components such as the header, tab bar etc. See theming guide for more details and usage guide.

`independent`

Whether this navigation container should be independent of parent containers. If this is not set to `true`, this container cannot be nested inside another container. Setting it to `true` disconnects any children navigators from parent container.

You probably don't want to set this to `true` in a typical React Native app. This is only useful if you have navigation trees that work like their own mini-apps and don't need to navigate to the screens outside of them.

Edit this page