Version: 6.x

Navigation prop reference

Each `screen` component in your app is provided with the `navigation` prop automatically. The prop contains various convenience functions that dispatch navigation actions. It looks like this:

- `navigation`
  - `navigate` - go to another screen, figures out the action it needs to take to do it
  - `reset` - wipe the navigator state and replace it with a new route
  - `goBack` - close active screen and move back in the stack
  - `setParams` - make changes to route's params
  - `dispatch` - send an action object to update the navigation state
  - `setOptions` - update the screen's options
  - `isFocused` - check whether the screen is focused
  - `addListener` - subscribe to updates to events from the navigators

It's important to highlight the `navigation` prop is *not* passed in to *all* components; only `screen` components receive this prop automatically! React Navigation doesn't do any magic here. For example, if you were to define a `MyBackButton` component and render it as a child of a screen component, you would not be able to access the `navigation` prop on it. If, however, you wish to access the `navigation` prop in any of your components, you may use the `useNavigation` hook.

> `setParams`/`setOptions` etc. should only be called in `useEffect`/`useLayoutEffect`/`componentDidMount`/`componentDidUpdate` etc. Not during render or in constructor.

Navigator-dependent functions

There are several additional functions present on `navigation` prop based on the kind of the current navigator.

If the navigator is a stack navigator, several alternatives to `navigate` and `goBack` are provided and you can use whichever you prefer. The functions are:

- `navigation`
  - `replace` - replace the current screen with a new one
  - `push` - push a new screen onto the stack
  - `pop` - go back in the stack
  - `popToTop` - go to the top of the stack

See Stack navigator helpers and Native Stack navigator helpers for more details on these methods.

If the navigator is a tab navigator, the following are also available:

- `navigation`
  - `jumpTo` - go to a specific screen in the tab navigator

See Bottom Tab navigator helpers, Material Top Tab navigator helpers and Material Bottom Tab navigator helpers for more details on these methods.

If the navigator is a drawer navigator, the following are also available:

- `navigation`
  - `jumpTo` - go to a specific screen in the drawer navigator
  - `openDrawer` - open the drawer
  - `closeDrawer` - close the drawer
  - `toggleDrawer` - toggle the state, ie. switch from closed to open and vice versa

See Drawer navigator helpers for more details on these methods.

Common API reference

The vast majority of your interactions with the `navigation` prop will involve `navigate`, `goBack`, and `setParams`.

`navigate`

The `navigate` method lets us navigate to another screen in your app. It takes the following arguments:

`navigation.navigate(name, params)`

- `name` - A destination name of the route that has been defined somewhere
- `params` - Params to pass to the destination route.

```
function HomeScreen({ navigation: { navigate } }) {
  return (
    <View>
      <Text>This is the home screen of the app</Text>
      <Button
        onPress={() =>
          navigate('Profile', { names: ['Brent', 'Satya', 'Michaś'] })
        }
        title="Go to Brent's profile"
      />
    </View>
  );
}
```

Try this example on Snack

In a native stack navigator, calling `navigate` with a screen name will result in different behavior based on if the screen is already present or not. If the screen is already present in the stack's history, it'll go back to that screen and remove any screens after that. If the screen is not present, it'll push a new screen.

For example, if you have a stack with the history `Home > Profile > Settings` and you call `navigate(Profile)`, the resulting screens will be `Home > Profile` as it goes back to `Profile` and removes the `Settings` screen.

By default, the screen is identified by its name. But you can also customize it to take the params into account by using the `getId` prop.

For example, say you have specified a `getId` prop for `Profile` screen:

```
<Tab.Screen name={Profile} component={ProfileScreen} getId={({ params }) =>
params.userId} />
```

Now, if you have a stack with the history `Home > Profile (userId: bob) > Settings` and you call `navigate(Profile, { userId: 'alice' })`, the resulting screens will be `Home > Profile` `(userId: bob) > Settings > Profile (userId: alice)` since it'll add a new `Profile` screen as no matching screen was found.

`goBack`

The `goBack` method lets us go back to the previous screen in the navigator.

By default, `goBack` will go back from the screen that it is called from:

```
function ProfileScreen({ navigation: { goBack } }) {
  return (
    <View>
      <Button onPress={() => goBack()} title="Go back from ProfileScreen" />
    </View>
  );
}
```

Try this example on Snack

Going back from a specific screen

Consider the following navigation stack history:

```
navigation.navigate({ name: SCREEN, key: SCREEN_KEY_A });
navigation.navigate({ name: SCREEN, key: SCREEN_KEY_B });
navigation.navigate({ name: SCREEN, key: SCREEN_KEY_C });
navigation.navigate({ name: SCREEN, key: SCREEN_KEY_D });
```

Now you are on *screen D* and want to go back to *screen A* (popping D, C, and B). Then you can use `navigate`:

```
navigation.navigate({ key: SCREEN_KEY_A }); // will go to screen A FROM screen D
```

Alternatively, as *screen A* is the top of the stack, you can use `navigation.popToTop()`.

`reset`

The `reset` method lets us replace the navigator state with a new state:

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

Try this example on Snack

The state object specified in `reset` replaces the existing navigation state with the new one, i.e. removes existing screens and add new ones. If you want to preserve the existing screens when changing the state, you can use `CommonActions.reset` with `dispatch` instead.

> 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.

`setParams`

The `setParams` method lets us update the params (`route.params`) of the current screen. `setParams` works like React's `setState` - it shallow merges the provided params object with the current params.

```
function ProfileScreen({ navigation: { setParams } }) {
  return (
    <Button
      onPress={() =>
        setParams({
          friends:
            route.params.friends[0] === 'Brent'
              ? ['Wojciech', 'Szymon', 'Jakub']
              : ['Brent', 'Satya', 'Michaś'],
          title:
            route.params.title === "Brent's Profile"
              ? "Lucy's Profile"
              : "Brent's Profile",
        })
      }
      title="Swap title and friends"
    />
  );
}
```

Try this example on Snack

`setOptions`

The `setOptions` method lets us set screen options from within the component. This is useful if we need to use the component's props, state or context to configure our screen.

```
function ProfileScreen({ navigation, route }) {
  const [value, onChangeText] = React.useState(route.params.title);

  React.useEffect(() => {
    navigation.setOptions({
      title: value === '' ? 'No title' : value,
    });
  }, [navigation, value]);

  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <TextInput
        style={{ height: 40, borderColor: 'gray', borderWidth: 1 }}
        onChangeText={onChangeText}
        value={value}
      />
      <Button title="Go back" onPress={() => navigation.goBack()} />
    </View>
  );
}
```

Try this example on Snack

Any options specified here are shallow merged with the options specified when defining the screen.

When using `navigation.setOptions`, we recommend specifying a placeholder in the screen's `options` prop and update it using `navigation.setOptions`. This makes sure that the delay for updating the options isn't noticeable to the user. It also makes it work with lazy-loaded screens.

You can also use `React.useLayoutEffect` to reduce the delay in updating the options. But we recommend against doing it if you support web and do server side rendering.

> Note: `navigation.setOptions` is intended to provide the ability to update existing options when necessary. It's not a replacement for the `options` prop on the screen. Make sure to use `navigation.setOptions` sparingly only when absolutely necessary.

Navigation events

Screens can add listeners on the `navigation` prop with the `addListener` method. For example, to listen to the `focus` event:

```
function Profile({ navigation }) {
  React.useEffect(() => {
    const unsubscribe = navigation.addListener('focus', () => {
      // do something
    });

    return unsubscribe;
  }, [navigation]);

  return <ProfileContent />;
}
```

Try this example on Snack

See Navigation events for more details on the available events and the API usage.

`isFocused`

This method lets us check whether the screen is currently focused. Returns `true` if the screen is focused and `false` otherwise.

```
const isFocused = navigation.isFocused();
```

This method doesn't re-render the screen when the value changes and mainly useful in callbacks. You probably want to use useIsFocused instead of using this directly, it will return a boolean a prop to indicating if the screen is focused.

Advanced API Reference

The `dispatch` function is much less commonly used, but a good escape hatch if you can't do what you need with the available methods such as `navigate`, `goBack` etc. We recommend to avoid using the `dispatch` method often unless absolutely necessary.

`dispatch`

The `dispatch` method lets us send a navigation action object which determines how the navigation state will be updated. All of the navigation functions like `navigate` use `dispatch` behind the scenes.

Note that if you want to dispatch actions you should use the action creators provided in this library instead of writing the action object directly.

See Navigation Actions Docs for a full list of available actions.

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

navigation.dispatch(
  CommonActions.navigate({
    name: 'Profile',
    params: {},
  })
);
```

When dispatching action objects, you can also specify few additional properties:

- `source` - The key of the route which should be considered as the source of the action. For example, the `replace` action will replace the route with the given key. By default, it'll use the key of the route that dispatched the action. You can explicitly pass `undefined` to override this behavior.
- `target` - The key of the navigation state the action should be applied on. By default, actions bubble to other navigators if not handled by a navigator. If `target` is specified, the action won't bubble if the navigator with the same key didn't handle it.

Example:

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

navigation.dispatch({
  ...CommonActions.navigate('Profile'),
  source: 'someRoutekey',
  target: 'someStatekey',
});
```

Custom action creators

It's also possible to pass a action creator function to `dispatch`. The function will receive the current state and needs to return a navigation action object to use:

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

navigation.dispatch((state) => {
  // Add the home route to the start of the stack
  const routes = [{ name: 'Home' }, ...state.routes];

  return CommonActions.reset({
    ...state,
    routes,
    index: routes.length - 1,
  });
});
```

You can use this functionality to build your own helpers that you can utilize in your app. Here is an example which implements inserting a screen just before the last one:

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

const insertBeforeLast = (routeName, params) => (state) => {
  const routes = [
    ...state.routes.slice(0, -1),
    { name: routeName, params },
    state.routes[state.routes.length - 1],
  ];
```

```
    return CommonActions.reset({
      ...state,
      routes,
      index: routes.length - 1,
    });
  };
```

Then use it like:

```
navigation.dispatch(insertBeforeLast('Home'));
```

`canGoBack`

This method returns a boolean indicating whether there's any navigation history available in the current navigator, or in any parent navigators. You can use this to check if you can call `navigation.goBack()`:

```
if (navigation.canGoBack()) {
  navigation.goBack();
}
```

Don't use this method for rendering content as this will not trigger a re-render. This is only intended for use inside callbacks, event listeners etc.

`getParent`

This method returns the navigation prop from the parent navigator that the current navigator is nested in. For example, if you have a stack navigator and a tab navigator nested inside the stack, then you can use `getParent` inside a screen of the tab navigator to get the navigation prop passed from the stack navigator.

It accepts an optional ID parameter to refer to a specific parent navigator. For example, if your screen is nested with multiple levels of nesting somewhere under a drawer navigator with the `id` prop as `"LeftDrawer"`, you can directly refer to it without calling `getParent` multiple times.

To use an ID for a navigator, first pass a unique `id` prop:

```
<Drawer.Navigator id="LeftDrawer">
  {/* .. */}
</Drawer.Navigator>
```

Then when using `getParent`, instead of:

```
// Avoid this
const drawerNavigation = navigation.getParent().getParent();

// ...

drawerNavigation?.openDrawer();
```

You can do:

```
// Do this
const drawerNavigation = navigation.getParent('LeftDrawer');

// ...

drawerNavigation?.openDrawer();
```

This approach allows components to not have to know the nesting structure of the navigators. So it's highly recommended that use an `id` when using `getParent`.

This method will return `undefined` if there is no matching parent navigator. Be sure to always check for `undefined` when using this method.

`getState`

> 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.

This method returns the state object of the navigator which contains the screen. Getting the navigator state could be useful in very rare situations. You most likely don't need to use this method.

If you do, make sure you have a good reason.

If you need the state for rendering content, you should use `useNavigationState` instead of this method.

Edit this page