



Version: 6.x

Custom navigators

Navigators allow you to define your application's navigation structure. Navigators also render common elements such as headers and tab bars which you can configure.

Under the hood, navigators are plain React components.

Built-in Navigators

We include some commonly needed navigators such as:

- `createStackNavigator` - Renders one screen at a time and provides transitions between screens. When a new screen is opened it is placed on top of the stack.
- `createDrawerNavigator` - Provides a drawer that slides in from the left of the screen by default.
- `createBottomTabNavigator` - Renders a tab bar that lets the user switch between several screens.
- `createMaterialTopTabNavigator` - Renders tab view which lets the user switch between several screens using swipe gesture or the tab bar.
- `createMaterialBottomTabNavigator` - Renders tab view which lets the user switch between several screens using swipe gesture or the tab bar.

API for building custom navigators

A navigator bundles a router and a view which takes the `navigation state` and decides how to render it. We export a `useNavigationBuilder` hook to build custom navigators that integrate with rest of React Navigation.

`useNavigationBuilder`

This hook allows a component to hook into React Navigation. It accepts the following arguments:

- `createRouter` - A factory method which returns a router object (e.g. `StackRouter`, `TabRouter`).
- `options` - Options for the hook and the router. The navigator should forward its props here so that user can provide props to configure the navigator. By default, the following options are accepted:
 - `children` (required) - The `children` prop should contain route configurations as `Screen` components.
 - `screenOptions` - The `screenOptions` prop should contain default options for all of the screens.
 - `initialRouteName` - The `initialRouteName` prop determines the screen to focus on initial render. This prop is forwarded to the router.

If any other options are passed here, they'll be forwarded to the router.

The hook returns an object with following properties:

- `state` - The navigation state for the navigator. The component can take this state and decide how to render it.
- `navigation` - The navigation object containing various helper methods for the navigator to manipulate the navigation state. This isn't the same as the navigation object for the screen and includes some helpers such as `emit` to emit events to the screens.
- `descriptors` - This is an object containing descriptors for each route with the route keys as its properties. The descriptor for a route can be accessed by `descriptors[route.key]`. Each descriptor contains the following properties:
 - `navigation` - The navigation prop for the screen. You don't need to pass this to the screen manually. But it's useful if we're rendering components outside the screen that need to receive `navigation` prop as well, such as a header component.
 - `options` - A getter which returns the options such as `title` for the screen if they are specified.
 - `render` - A function which can be used to render the actual screen. Calling `descriptors[route.key].render()` will return a React element containing the screen

content. It's important to use this method to render a screen, otherwise any child navigators won't be connected to the navigation tree properly.

Example:

```
import * as React from 'react';
import { Text, Pressable, View } from 'react-native';
import {
  NavigationHelpersContext,
  useNavigationBuilder,
  TabRouter,
  TabActions,
} from '@react-navigation/native';

function TabNavigator({
  initialRouteName,
  children,
  screenOptions,
  tabBarStyle,
  contentStyle,
}) {
  const { state, navigation, descriptors, NavigationContent } =
    useNavigationBuilder(TabRouter, {
      children,
      screenOptions,
      initialRouteName,
    });

  return (
    <NavigationContent>
      <View style={[{ flexDirection: 'row' }, tabBarStyle]>
        {state.routes.map((route) => (
          <Pressable
            key={route.key}
            onPress={() => {
              const event = navigation.emit({
                type: 'tabPress',
                target: route.key,
                canPreventDefault: true,
              });

              if (!event.defaultPrevented) {
                navigation.dispatch({
```

```

        ...TabActions.jumpTo(route.name),
        target: state.key,
      });
    }
  }
  style={{ flex: 1 }}
>
  <Text>{descriptors[route.key].options.title || route.name}</Text>
</Pressable>
)))
</View>
<View style={[{ flex: 1 }, contentStyle]}>
  {state.routes.map((route, i) => {
    return (
      <View
        key={route.key}
        style={[
          StyleSheet.absoluteFill,
          { display: i === state.index ? 'flex' : 'none' },
        ]}
      >
        {descriptors[route.key].render()}
      </View>
    );
  })}
</View>
</NavigationContent>
);
}

```

The `navigation` object for navigators also has an `emit` method to emit custom events to the child screens. The usage looks like this:

```

navigation.emit({
  type: 'transitionStart',
  data: { blurring: false },
  target: route.key,
});

```

The `data` is available under the `data` property in the `event` object, i.e. `event.data`.

The `target` property determines the screen that will receive the event. If the `target` property is omitted, the event is dispatched to all screens in the navigator.

`createNavigatorFactory`

This `createNavigatorFactory` function is used to create a function that will `Navigator` and `Screen` pair. Custom navigators need to wrap the navigator component in `createNavigatorFactory` before exporting.

Example:

```
import {
  useNavigationBuilder,
  createNavigatorFactory,
} from '@react-navigation/native';

// ...

export const createMyNavigator = createNavigatorFactory(TabNavigator);
```

Then it can be used like this:

```
import { createMyNavigator } from './myNavigator';

const My = createMyNavigator();

function App() {
  return (
    <My.Navigator>
      <My.Screen name="Home" component={HomeScreen} />
      <My.Screen name="Feed" component={FeedScreen} />
    </My.Navigator>
  );
}
```

Type-checking navigators

To type-check navigators, we need to provide 3 types:

- Type of the props accepted by the view
- Type of supported screen options
- A map of event types emitted by the navigator

For example, to type-check our custom tab navigator, we can do something like this:

```
import * as React from 'react';
import {
  View,
  Text,
  Pressable,
  StyleProp,
  ViewStyle,
  StyleSheet,
} from 'react-native';
import {
  createNavigatorFactory,
  DefaultNavigatorOptions,
  ParamListBase,
  CommonActions,
  TabActionHelpers,
  TabNavigationState,
  TabRouter,
  TabRouterOptions,
  useNavigationBuilder,
} from '@react-navigation/native';

// Props accepted by the view
type TabNavigationConfig = {
  tabBarStyle: StyleProp<ViewStyle>;
  contentStyle: StyleProp<ViewStyle>;
};

// Supported screen options
type TabNavigationOptions = {
  title?: string;
};

// Map of event name and the type of data (in event.data)
//
// canPreventDefault: true adds the defaultPrevented property to the
// emitted events.
```

```

type TabNavigationEventMap = {
  tabPress: {
    data: { isAlreadyFocused: boolean };
    canPreventDefault: true;
  };
};

// The props accepted by the component is a combination of 3 things
type Props = DefaultNavigatorOptions<
  ParamListBase,
  TabNavigationState<ParamListBase>,
  TabNavigationOptions,
  TabNavigationEventMap
> &
  TabRouterOptions &
  TabNavigationConfig;

function TabNavigator({
  initialRouteName,
  children,
  screenOptions,
  tabBarStyle,
  contentStyle,
}: Props) {
  const { state, navigation, descriptors, NavigationContent } =
    useNavigationBuilder<
      TabNavigationState<ParamListBase>,
      TabRouterOptions,
      TabActionHelpers<ParamListBase>,
      TabNavigationOptions,
      TabNavigationEventMap
    >(TabRouter, {
      children,
      screenOptions,
      initialRouteName,
    });

  return (
    <NavigationContent>
      <View style={[{ flexDirection: 'row' }, tabBarStyle]}>
        {state.routes.map((route) => (
          <Pressable
            key={route.key}
            onPress={() => {

```

```

    const event = navigation.emit({
      type: 'tabPress',
      target: route.key,
      canPreventDefault: true,
      data: {
        isAlreadyFocused: route.key === state.routes[state.index].key,
      },
    });

    if (!event.defaultPrevented) {
      navigation.dispatch({
        ...CommonActions.navigate(route),
        target: state.key,
      });
    }
  }}
  style={{ flex: 1 }}
>
  <Text>{descriptors[route.key].options.title || route.name}</Text>
</Pressable>
)))}
</View>
<View style={[{ flex: 1 }, contentType]}>
  {state.routes.map((route, i) => {
    return (
      <View
        key={route.key}
        style={[
          StyleSheet.absoluteFill,
          { display: i === state.index ? 'flex' : 'none' },
        ]}
      >
        {descriptors[route.key].render()}
      </View>
    );
  })}
</View>
</NavigationContent>
);
}

export default createNavigatorFactory<
  TabNavigationState<ParamListBase>,
  TabNavigationOptions,

```



```
TabNavigationEventMap,  
  typeof TabNavigator  
>(TabNavigator);
```

Extending Navigators

All of the built-in navigators export their views, which we can reuse and build additional functionality on top of them. For example, if we want to re-build the bottom tab navigator, we need the following code:

```
import * as React from 'react';  
import {  
  useNavigationBuilder,  
  createNavigatorFactory,  
  TabRouter,  
} from '@react-navigation/native';  
import { BottomTabView } from '@react-navigation/bottom-tabs';  
  
function BottomTabNavigator({  
  initialRouteName,  
  backBehavior,  
  children,  
  screenOptions,  
  ...rest  
) {  
  const { state, descriptors, navigation, NavigationContent } =  
    useNavigationBuilder(TabRouter, {  
      initialRouteName,  
      backBehavior,  
      children,  
      screenOptions,  
    });  
  
  return (  
    <NavigationContent>  
      <BottomTabView  
        {...rest}  
        state={state}  
        navigation={navigation}  
        descriptors={descriptors}  
      />
```

```
    </NavigationContent>
  );
}

export default createNavigatorFactory(BottomTabNavigator);
```


Now, we can customize it to add additional functionality or change the behavior. For example, use a custom router instead of the default `TabRouter`:

```
import MyRouter from './MyRouter';

// ...

const { state, descriptors, navigation, NavigationContent } =
  useNavigationBuilder(MyRouter, {
    initialRouteName,
    backBehavior,
    children,
    screenOptions,
  });

// ...
```

 [Edit this page](#)