



Guides

Configuring links

Version: 6.x

Configuring links

In this guide, we will configure React Navigation to handle external links. This is necessary if you want to:

1. Handle deep links in React Native apps on Android and iOS
2. Enable URL integration in browser when using on web
3. Use `<Link />` or `useLinkTo` to navigate using paths.

Make sure that you have configured deep links in your app before proceeding. If you have an Android or iOS app, remember to specify the `prefixes` option.

The `NavigationContainer` accepts a `linking` prop that makes it easier to handle incoming links. The 2 of the most important properties you can specify in the `linking` prop are `prefixes` and `config`:

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

const linking = {
  prefixes: [
    /* your linking prefixes */
  ],
  config: {
    /* configuration for matching screens with paths */
  },
};

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

When you specify the `linking` prop, React Navigation will handle incoming links automatically. On Android and iOS, it'll use React Native's `Linking` module to handle incoming links, both when the app was opened with the link, and when new links are received when the app is open. On the Web, it'll use the `History API` to sync the URL with the browser.

Note: Currently there seems to be bug ([facebook/react-native#25675](https://github.com/facebook/react-native/issues/25675)) which results in it never resolving on Android. We add a timeout to avoid getting stuck forever, but it means that the link might not be handled in some cases.

You can also pass a `fallback` prop to `NavigationContainer` which controls what's displayed when React Navigation is trying to resolve the initial deep link URL.

Prefixes

The `prefixes` option can be used to specify custom schemes (e.g. `mychat://`) as well as host & domain names (e.g. `https://mychat.com`) if you have configured `Universal Links` or `Android App Links`.

For example:

```
const linking = {
  prefixes: ['mychat://', 'https://mychat.com'],
};
```

Note that the `prefixes` option is not supported on Web. The host & domain names will be automatically determined from the Website URL in the browser. If your app runs only on Web, then you can omit this option from the config.

Multiple subdomains

To match all subdomains of an associated domain, you can specify a wildcard by prefixing `*` before the beginning of a specific domain. Note that an entry for `*.mychat.com` does not match `mychat.com` because of the period after the asterisk. To enable matching for both `*.mychat.com` and `mychat.com`, you need to provide a separate prefix entry for each.

```
const linking = {
  prefixes: ['mychat://', 'https://mychat.com', 'https://*.mychat.com'],
};
```

```
};
```

Mapping path to route names

To handle a link, you need to translate it to a valid [navigation state](#) and vice versa. For example, the path `/rooms/chat?user=jane` may be translated to a state object like this:

```
const state = {
  routes: [
    {
      name: 'rooms',
      state: {
        routes: [
          {
            name: 'chat',
            params: { user: 'jane' },
          },
        ],
      },
    },
  ],
};
```

By default, React Navigation will use the path segments as the route name when parsing the URL. But directly translating path segments to route names may not be the expected behavior.

For example, you might want to parse the path `/feed/latest` to something like:

```
const state = {
  routes: [
    {
      name: 'Chat',
      params: {
        sort: 'latest',
      },
    },
  ],
};
```

You can specify the `config` option in `linking` to control how the deep link is parsed to suit your needs.

```
const config = {
  screens: {
    Chat: 'feed/:sort',
    Profile: 'user',
  },
};
```

Here `Chat` is the name of the screen that handles the URL `/feed`, and `Profile` handles the URL `/user`.

The config option can then be passed in the `linking` prop to the container:

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

const config = {
  screens: {
    Chat: 'feed/:sort',
    Profile: 'user',
  },
};

const linking = {
  prefixes: ['https://mychat.com', 'mychat:///'],
  config,
};

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

The config object must match the navigation structure for your app. For example, the above configuration is if you have `Chat` and `Profile` screens in the navigator at the root:

```
function App() {  
  return (  
    <Stack.Navigator>  
      <Stack.Screen name="Chat" component={ChatScreen} />  
      <Stack.Screen name="Profile" component={ProfileScreen} />  
    </Stack.Navigator>  
  );  
}
```

If your `Chat` screen is inside a nested navigator, we'd need to account for that. For example, consider the following structure where your `Profile` screen is at the root, but the `Chat` screen is nested inside `Home`:

```
function App() {  
  return (  
    <Stack.Navigator>  
      <Stack.Screen name="Home" component={HomeScreen} />  
      <Stack.Screen name="Profile" component={ProfileScreen} />  
    </Stack.Navigator>  
  );  
}  
  
function HomeScreen() {  
  return (  
    <Tab.Navigator>  
      <Tab.Screen name="Chat" component={ChatScreen} />  
    </Tab.Navigator>  
  );  
}
```

For above structure, our configuration will look like this:

```
const config = {  
  screens: {  
    Home: {  
      screens: {  
        Chat: 'feed/:sort',  
      },  
    },  
    Profile: 'user',  
  },  
}
```

```
    },  
  };  
};
```

Similarly, any nesting needs to be reflected in the configuration. See [handling nested navigators](#) for more details.

Passing params

A common use case is to pass params to a screen to pass some data. For example, you may want the `Profile` screen to have an `id` param to know which user's profile it is. It's possible to pass params to a screen through a URL when handling deep links.

By default, query params are parsed to get the params for a screen. For example, with the above example, the URL `/user?id=wojciech` will pass the `id` param to the `Profile` screen.

You can also customize how the params are parsed from the URL. Let's say you want the URL to look like `/user/wojciech` where the `id` param is `wojciech` instead of having the `id` in query params. You can do this by specifying `user/:id` for the `path`. **When the path segment starts with `:`, it'll be treated as a param.** For example, the URL `/user/wojciech` would resolve to `Profile` screen with the string `wojciech` as a value of the `id` param and will be available in `route.params.id` in `Profile` screen.

By default, all params are treated as strings. You can also customize how to parse them by specifying a function in the `parse` property to parse the param, and a function in the `stringify` property to convert it back to a string.

If you wanted to resolve `/user/wojciech/settings` to result in the params `{ id: 'user-wojciech' section: 'settings' }`, you could make `Profile`'s config to look like this:

```
const config = {  
  screens: {  
    Profile: {  
      path: 'user/:id/:section',  
      parse: {  
        id: (id) => `user-${id}`,  
      },  
      stringify: {  
        id: (id) => id.replace(/^user-/ , ''),  
      },  
    },  
  },  
};
```

```
    },  
  },  
},  
};
```

This will result in something like:

```
const state = {  
  routes: [  
    {  
      name: 'Profile',  
      params: { id: 'user-wojciech', section: 'settings' },  
    },  
  ],  
};
```

Marking params as optional

Sometimes a param may or may not be present in the URL depending on certain conditions. For example, in the above scenario, you may not always have the section parameter in the URL, i.e. both `/user/wojciech/settings` and `/user/wojciech` should go to the `Profile` screen, but the `section` param (with the value `settings` in this case) may or may not be present.

In this case, you would need to mark the `section` param as optional. You can do it by adding the `?` suffix after the param name:

```
const config = {  
  screens: {  
    Profile: {  
      path: 'user/:id/:section?',  
      parse: {  
        id: (id) => `user-${id}`,  
      },  
      stringify: {  
        id: (id) => id.replace(/^user-/ , ''),  
      },  
    },  
  },  
};
```

```
    },  
  };  
};
```

With the URL `/users/wojciech`, this will result in:

```
const state = {  
  routes: [  
    {  
      name: 'Profile',  
      params: { id: 'user-wojciech' },  
    },  
  ],  
};
```

If the URL contains a `section` param, e.g. `/users/wojciech/settings`, this will result in the following with the same config:

```
const state = {  
  routes: [  
    {  
      name: 'Profile',  
      params: { id: 'user-wojciech', section: 'settings' },  
    },  
  ],  
};
```

Handling nested navigators

Sometimes you'll have the target navigator nested in other navigators which aren't part of the deep link. For example, let's say your navigation structure looks like this:

```
function Home() {  
  return (  
    <Tab.Navigator>  
      <Tab.Screen name="Profile" component={Profile} />  
      <Tab.Screen name="Feed" component={Feed} />  
    </Tab.Navigator>  
  );  
};
```



```
}

function App() {
  return (
    <Stack.Navigator>
      <Stack.Screen name="Home" component={Home} />
      <Stack.Screen name="Settings" component={Settings} />
    </Stack.Navigator>
  );
}
```

Here you have a stack navigator in the root, and inside the `Home` screen of the root stack, you have a tab navigator with various screens. With this structure, let's say you want the path `/users/:id` to go to the `Profile` screen. You can express the nested config like so:

```
const config = {
  screens: {
    Home: {
      screens: {
        Profile: 'users/:id',
      },
    },
  },
};
```

In this config, you specify that the `Profile` screen should be resolved for the `users/:id` pattern and it's nested inside the `Home` screen. Then parsing `users/jane` will result in the following state object:

```
const state = {
  routes: [
    {
      name: 'Home',
      state: {
        routes: [
          {
            name: 'Profile',
            params: { id: 'jane' },
          },
        ],
      },
    },
  ],
};
```

```
    },  
  },  
],  
};
```

It's important to note that the state object must match the hierarchy of nested navigators. Otherwise the state will be discarded.

Handling unmatched routes or 404

If your app is opened with an invalid URL, most of the times you'd want to show an error page with some information. On the web, this is commonly known as 404 - or page not found error.

To handle this, you'll need to define a catch-all route that will be rendered if no other routes match the path. You can do it by specifying `*` for the path matching pattern.

For example:

```
const config = {  
  screens: {  
    Home: {  
      initialRouteName: 'Feed',  
      screens: {  
        Profile: 'users/:id',  
        Settings: 'settings',  
      },  
    },  
  },  
  NotFound: '*',  
};
```

Here, we have defined a route named `NotFound` and set it to match `*` aka everything. If the path didn't match `user/:id` or `settings`, it'll be matched by this route.

So, a path like `/library` or `/settings/notification` will resolve to the following state object:

```
const state = {  
  routes: [{ name: 'NotFound' }],
```

```
};
```

You can even go more specific, for example, say if you want to show a different screen for invalid paths under `/settings`, you can specify such a pattern under `Settings`:

```
const config = {
  screens: {
    Home: {
      initialRouteName: 'Feed',
      screens: {
        Profile: 'users/:id',
        Settings: {
          path: 'settings',
          screens: {
            InvalidSettings: '*',
          },
        },
      },
    },
    NotFound: '*',
  },
};
```

With this configuration, the path `/settings/notification` will resolve to the following state object:

```
const state = {
  routes: [
    {
      name: 'Home',
      state: {
        index: 1,
        routes: [
          { name: 'Feed' },
          {
            name: 'Settings',
            state: {
              routes: [
                { name: 'InvalidSettings', path: '/settings/notification' },
              ],
            },
          },
        ],
      },
    },
  ],
};
```

```
    },  
  ],  
},  
},  
],  
};
```

The `route` passed to the `NotFound` screen will contain a `path` property which corresponds to the path that opened the page. If you need, you can use this property to customize what's shown in this screen, e.g. load the page in a `WebView`:

```
function NotFoundScreen({ route }) {  
  if (route.path) {  
    return <WebView source={{ uri: `https://mywebsite.com/${route.path}` }} />;  
  }  
  
  return <Text>This screen doesn't exist!</Text>;  
}
```

When doing server rendering, you'd also want to return correct status code for 404 errors. See [server rendering docs](#) for a guide on how to handle it.

Rendering an initial route

Sometimes you want to ensure that a certain screen will always be present as the first screen in the navigator's state. You can use the `initialRouteName` property to specify the screen to use for the initial screen.

In the above example, if you want the `Feed` screen to be the initial route in the navigator under `Home`, your config will look like this:

```
const config = {  
  screens: {  
    Home: {  
      initialRouteName: 'Feed',  
      screens: {  
        Profile: 'users/:id',  
        Settings: 'settings',  
      },  
    },  
  },  
};
```

```
    },  
  },  
},  
};
```

Then, the path `/users/42` will resolve to the following state object:

```
const state = {  
  routes: [  
    {  
      name: 'Home',  
      state: {  
        index: 1,  
        routes: [  
          { name: 'Feed' },  
          {  
            name: 'Profile',  
            params: { id: '42' },  
          },  
        ],  
      },  
    ],  
  },  
],  
};
```

It's not possible to pass params to the initial screen through the URL. So make sure that your initial route doesn't need any params or specify `initialParams` to pass required params.

In this case, any params in the URL are only passed to the `Profile` screen which matches the path pattern `users/:id`, and the `Feed` screen doesn't receive any params. If you want to have the same params in the `Feed` screen, you can specify a custom `getStateFromPath` function and copy those params.

Similarly, if you want to access params of a parent screen from a child screen, you can use `React Context` to expose them.

Matching exact paths

By default, paths defined for each screen are matched against the URL relative to their parent screen's path. Consider the following config:

```
const config = {
  screens: {
    Home: {
      path: 'feed',
      screens: {
        Profile: 'users/:id',
      },
    },
  },
};
```

Here, you have a `path` property defined for the `Home` screen, as well as the child `Profile` screen. The profile screen specifies the path `users/:id`, but since it's nested inside a screen with the path `feed`, it'll try to match the pattern `feed/users/:id`.

This will result in the URL `/feed` navigating to `Home` screen, and `/feed/users/cal` navigating to the `Profile` screen.

In this case, it makes more sense to navigate to the `Profile` screen using a URL like `/users/cal`, rather than `/feed/users/cal`. To achieve this, you can override the relative matching behavior to `exact` matching:

```
const config = {
  screens: {
    Home: {
      path: 'feed',
      screens: {
        Profile: {
          path: 'users/:id',
          exact: true,
        },
      },
    },
  },
};
```

With `exact` property set to `true`, `Profile` will ignore the parent screen's `path` config and you'll be able to navigate to `Profile` using a URL like `users/cal`.

Omitting a screen from path

Sometimes, you may not want to have the route name of a screen in the path. For example, let's say you have a `Home` screen and our navigation state looks like this:

```
const state = {  
  routes: [{ name: 'Home' }],  
};
```

When this state is serialized to a path with the following config, you'll get `/home`:

```
const config = {  
  screens: {  
    Home: {  
      path: 'home',  
      screens: {  
        Profile: 'users/:id',  
      },  
    },  
  },  
};
```

But it'll be nicer if the URL was just `/` when visiting the home screen. You can specify an empty string as path or not specify a path at all, and React Navigation won't add the screen to the path (think of it like adding empty string to the path, which doesn't change anything):

```
const config = {  
  screens: {  
    Home: {  
      path: '',  
      screens: {  
        Profile: 'users/:id',  
      },  
    },  
  },  
};
```

```
    },  
  };  
};
```

Serializing and parsing params

Since URLs are strings, any params you have for routes are also converted to strings when constructing the path.

For example, say you have a state like following:

```
const state = {  
  routes: [  
    {  
      name: 'Chat',  
      params: { at: 1589842744264 },  
    },  
  ],  
};
```

It'll be converted to `chat/1589842744264` with the following config:

```
const config = {  
  screens: {  
    Chat: 'chat/:date',  
  },  
};
```

When parsing this path, you'll get the following state:

```
const state = {  
  routes: [  
    {  
      name: 'Chat',  
      params: { date: '1589842744264' },  
    },  
  ],  
};
```


Here, the `date` param was parsed as a string because React Navigation doesn't know that it's supposed to be a timestamp, and hence number. You can customize it by providing a custom function to use for parsing:

```
const config = {
  screens: {
    Chat: {
      path: 'chat/:date',
      parse: {
        date: Number,
      },
    },
  },
};
```

You can also provide a custom function to serialize the params. For example, let's say that you want to use a DD-MM-YYYY format in the path instead of a timestamp:

```
const config = {
  screens: {
    Chat: {
      path: 'chat/:date',
      parse: {
        date: (date) => new Date(date).getTime(),
      },
      stringify: {
        date: (date) => {
          const d = new Date(date);

          return d.getFullYear() + '-' + d.getMonth() + '-' + d.getDate();
        },
      },
    },
  },
};
```

Depending on your requirements, you can use this functionality to parse and stringify more complex data.

Advanced cases

For some advanced cases, specifying the mapping may not be sufficient. To handle such cases, you can specify a custom function to parse the URL into a state object (`getStateFromPath`), and a custom function to serialize the state object into an URL (`getPathFromState`).

Example:

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

Updating config

Older versions of React Navigation had a slightly different configuration format for linking. The old config allowed a simple key value pair in the object regardless of nesting of navigators:

```
const config = {
  Home: 'home',
  Feed: 'feed',
  Profile: 'profile',
  Settings: 'settings',
};
```

Let's say, your `Feed` and `Profile` screens are nested inside `Home`. Even if you don't have such a nesting with the above configuration, as long as the URL was `/home/profile`, it would work. Furthermore, it would also treat path segments and route names the same, which means that you could deep link to a screen that's not specified in the configuration. For example, if you have a `Albums` screen inside `Home`, the deep link `/home/Albums` would navigate to that screen. While that may be desirable in some cases, there's no way to prevent access to specific screens. This approach also makes it impossible to have something like a 404 screen since any route name is a valid path.

Latest versions of React Navigation use a different config format which is stricter in this regard:

- The shape of the config must match the shape of the nesting in the navigation structure
- Only screens defined in the config will be eligible for deep linking

So, you'd refactor the above config to the following format:

```
const config = {
  screens: {
    Home: {
      path: 'home',
      screens: {
        Feed: 'feed',
        Profile: 'profile',
      },
    },
    Settings: 'settings',
  },
};
```

Here, there's a new `screens` property to the configuration object, and the `Feed` and `Profile` configs are now nested under `Home` to match the navigation structure.

If you have the old format, it will continue to work without any changes. However, you won't be able to specify a wildcard pattern to handle unmatched screens or prevent screens from being deep linked. The old format will be removed in the next major release. So we recommend to migrate to the new format when you can.

Playground

You can play around with customizing the config and path below, and see how the path is parsed.

/user/@vergil/edit

```
{
  screens: {
    Home: {
      initialRouteName: 'Feed',
      screens: {
        Profile: {
          path: 'user/:id',
          parse: {
            id: id => id.replace(/^@/, ''),
          },
          screens: {
            Settings: 'edit',
          },
        },
      },
    },
  },
  NoMatch: '*',
}
```

Chart State Action

Home

Feed

Profile

Settings

id : "vergil"

Example App

In the example app, you will use the Expo managed workflow. The guide will focus on creating the deep linking configuration and not on creating the components themselves, but you can always

check the full implementation in the [github repo](#).

First, you need to decide the navigation structure of your app. To keep it simple, the main navigator will be bottom-tabs navigator with two screens. Its first screen will be a simple stack navigator, called `HomeStack`, with two screens: `Home` and `Profile`, and the second tabs screen will be just a simple one without any nested navigators, called `Settings`:

```
BottomTabs
├── Stack (HomeStack)
│   ├── Home
│   └── Profile
└── Settings
```

After creating the navigation structure, you can create a config for deep linking, which will contain mappings for each screen to a path segment. For example:


```
const config = {
  screens: {
    HomeStack: {
      screens: {
        Home: 'home',
        Profile: 'user',
      },
    },
    Settings: 'settings',
  },
};
```

As you can see, `Home` and `Profile` are nested in the `screens` property of `HomeStack`. This means that when you pass the `/home` URL, it will be resolved to a `HomeStack` -> `Home` state object (similarly for `/user` it would be `HomeStack` -> `Profile`). The nesting in this object should match the nesting of our navigators.

Here, the `HomeStack` property contains a config object. The config can go as deep as you want, e.g. if `Home` was a navigator, you could make it an object with `screens` property, and put more screens or navigators inside it, making the URL string much more readable.

What if you wanted a specific screen to be used as the initial screen in the navigator? For example, if you had a URL that would open `Home` screen, you would like to be able to navigate to `Profile` from it by using navigation's `navigation.goBack()` method. It is possible by defining `initialRouteName` for a navigator. It would look like this:

```
const config = {
  screens: {
    HomeStack: {
      initialRouteName: 'Profile',
      screens: {
        Home: 'home',
        Profile: 'user',
      },
    },
    Settings: 'settings',
  },
};
```

 [Edit this page](#)