

[Guides](#)[Deep linking](#)

Version: 6.x

Deep linking

This guide will describe how to configure your app to handle deep links on various platforms. To handle incoming links, you need to handle 2 scenarios:

1. If the app wasn't previously open, the deep link needs to set the initial state
2. If the app was already open, the deep link needs to update the state to reflect the incoming link

React Native provides a `Linking` to get notified of incoming links. React Navigation can integrate with the `Linking` module to automatically handle deep links. On Web, React Navigation can integrate with browser's `history` API to handle URLs on client side. See [configuring links](#) to see more details on how to configure links in React Navigation.

While you don't need to use the `linking` prop from React Navigation, and can handle deep links yourself by using the `Linking` API and navigating from there, it'll be significantly more complicated than using the `linking` prop which handles many edge cases for you. So we don't recommend implementing it by yourself.

Below, we'll go through required configurations so that the deep link integration works.

Setup with Expo projects

First, you will want to specify a URL scheme for your app. This corresponds to the string before `://` in a URL, so if your scheme is `mychat` then a link to your app would be `mychat://`. You can register for a scheme in your `app.json` by adding a string under the scheme key:

```
{
  "expo": {
    "scheme": "mychat"
  }
}
```

Next, install `expo-linking` which we'd need to get the deep link prefix:

```
npx expo install expo-linking
```

Then, let's configure React Navigation to use the `scheme` for parsing incoming deep links:

```
import * as Linking from 'expo-linking';

const prefix = Linking.createURL('/');

function App() {
  const linking = {
    prefixes: [prefix],
  };

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

The reason that is necessary to use `Linking.createURL` is that the scheme will differ depending on whether you're in the client app or in a standalone app.

The scheme specified in `app.json` only applies to standalone apps. In the Expo client app you can deep link using `exp://ADDRESS:PORT/--/` where `ADDRESS` is often `127.0.0.1` and `PORT` is often `19000` - the URL is printed when you run `expo start`. The `Linking.createURL` function abstracts it out so that you don't need to specify them manually.

If you are using universal links, you need to add your domain to the prefixes as well:

```
const linking = {
  prefixes: [Linking.createURL('/'), 'https://app.example.com'],
};
```

Set up with bare React Native projects

Setup on iOS

Let's configure the native iOS app to open based on the `mychat://` URI scheme.

You'll need to link `RCTLinking` to your project by following the steps described here. To be able to listen to incoming app links, you'll need to add the following lines to `AppDelegate.m` in your project:

```
// Add the header at the top of the file:
#import <React/RCTLinkingManager.h>

// Add this inside `@implementation AppDelegate` above `@end`:
- (BOOL)application:(UIApplication *)application
    openURL:(NSURL *)url
    options:(NSDictionary<UIApplicationOpenURLOptionsKey,id> *)options
{
    return [RCTLinkingManager application:application openURL:url
    options:options];
}
```

If your app is using `Universal Links`, you'll need to add the following code as well:

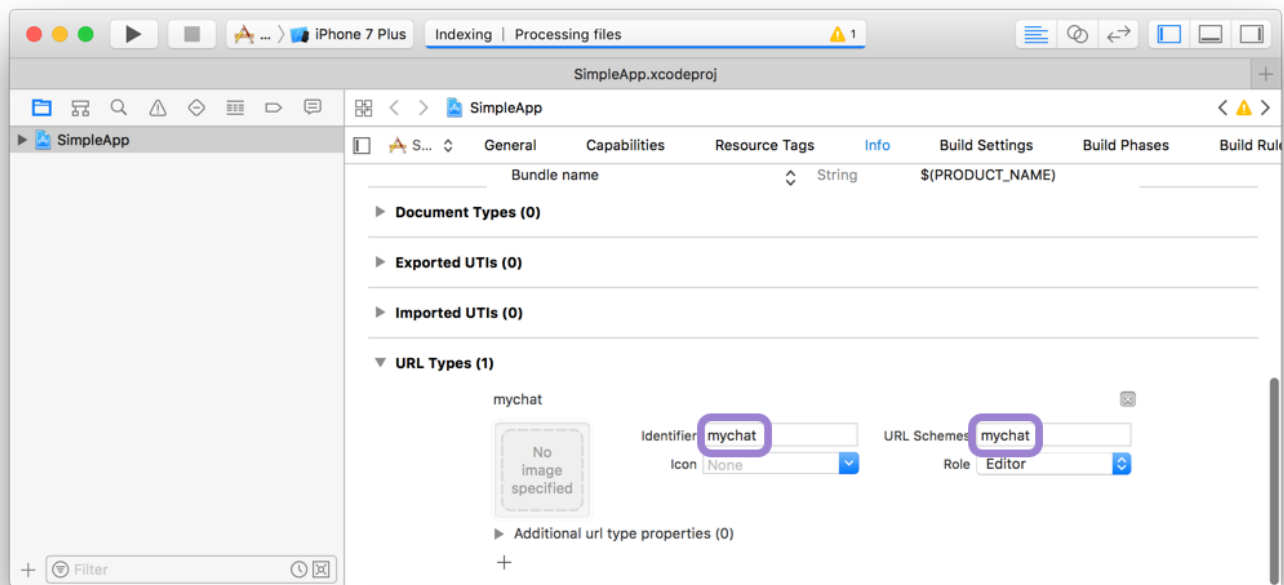
```
// Add this inside `@implementation AppDelegate` above `@end`:
- (BOOL)application:(UIApplication *)application continueUserActivity:(nonnull
    NSUserActivity *)userActivity
    restorationHandler:(nonnull void (^)(NSArray<id<UIUserActivityRestoring>> *
    _Nullable))restorationHandler
{
    return [RCTLinkingManager application:application
        continueUserActivity:userActivity
        restorationHandler:restorationHandler];
}
```

Now you need to add the scheme to your project configuration.

The easiest way to do this is with the `uri-scheme` package by running the following:

```
npx uri-scheme add mychat --ios
```

If you want to do it manually, open the project (e.g. `SimpleApp/ios/SimpleApp.xcworkspace`) in Xcode. Select the project in sidebar and navigate to the info tab. Scroll down to "URL Types" and add one. In the new URL type, set the identifier and the URL scheme to your desired URL scheme.



To make sure Universal Links work in your app, you also need to setup [Associated Domains](#) on your server.

Hybrid React Native and native iOS Applications

If you're using React Navigation within a hybrid app - an iOS app that has both Swift/ObjC and React Native parts - you may be missing the `RCTLinkingIOS` subspec in your `Podfile`, which is installed by default in new React Native projects. To add this, ensure your `Podfile` looks like the following:

```
pod 'React', :path => '../node_modules/react-native', :subspecs => [
  . . . // other subspecs
  'RCTLinkingIOS',
  . . .
]
```

Setup on Android

To configure the external linking in Android, you can create a new intent in the manifest.

The easiest way to do this is with the `uri-scheme` package: `npx uri-scheme add mychat --android`.

If you want to add it manually, open up

`SimpleApp/android/app/src/main/AndroidManifest.xml`, and make the following adjustments:

1. Set `launchMode` of `MainActivity` to `singleTask` in order to receive intent on existing `MainActivity` (this is the default, so you may not need to actually change anything).
2. Add the new `intent-filter` inside the `MainActivity` entry with a `VIEW` type action:

```
<activity
  android:name=".MainActivity"
  android:launchMode="singleTask">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="mychat" />
  </intent-filter>
</activity>
```

Similar to Universal Links on iOS, you can also use a domain to associate the app with your website on Android by verifying Android App Links. First, you need to configure your

`AndroidManifest.xml`:

1. Add `android:autoVerify="true"` to your `<intent-filter>` entry.
2. Add your domain's `scheme` and `host` in a new `<data>` entry inside the `<intent-filter>`.

After adding them, it should look like this:

```
<activity
  android:name=".MainActivity"
  android:launchMode="singleTask">
  <intent-filter android:autoVerify="true">
    <action android:name="android.intent.action.MAIN" />
```

```
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="mychat" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="http" />
        <data android:scheme="https" />
        <data android:host="www.example.com" />
    </intent-filter>
</activity>
```

Then, you need to declare the association between your website and your intent filters by hosting a Digital Asset Links JSON file.

Testing deep links

Before testing deep links, make sure that you rebuild and install the app in your emulator/simulator/device.

If you're testing on iOS, run:

```
npx react-native run-ios
```

If you're testing on Android, run:

```
npx react-native run-android
```

If you're using Expo managed workflow and testing on Expo client, you don't need to rebuild the app. However, you will need to use the correct address and port that's printed when you run `expo start` (see above), e.g. `exp://127.0.0.1:19000/--/`.

If you want to test with your custom scheme in your Expo app, you will need rebuild your standalone app by running `expo build:ios -t simulator` or `expo build:android` and install the resulting binaries.

Testing with `npx uri-scheme`

The `uri-scheme` package is a command line tool that can be used to test deep links on both iOS & Android. It can be used as follows:

```
npx uri-scheme open [your deep link] --[ios|android]
```

For example:

```
npx uri-scheme open "mychat://chat/jane" --ios
```

Or if using Expo client:

```
npx uri-scheme open "exp://127.0.0.1:19000/--/chat/jane" --ios
```

Testing with `xcrun` on iOS

The `xcrun` command can be used as follows to test deep links with the iOS simulator:

```
xcrun simctl openurl booted [your deep link]
```

For example:

```
xcrun simctl openurl booted "mychat://chat/jane"
```

Testing with `adb` on Android

The `adb` command can be used as follows to test deep links with the Android emulator or a connected device:

```
adb shell am start -W -a android.intent.action.VIEW -d [your deep link] [your android package name]
```

For example:

```
adb shell am start -W -a android.intent.action.VIEW -d "mychat://chat/jane" com.simpleapp
```

Or if using Expo client:

```
adb shell am start -W -a android.intent.action.VIEW -d "exp://127.0.0.1:19000/-/chat/jane" host.exp.exponent
```

Third-party integrations

React Native's `Linking` isn't the only way to handle deep linking. You might also want to integrate other services such as [Firebase Dynamic Links](#), [Branch](#) etc. which provide their own API for getting notified of incoming links.

To achieve this, you'd need to override how React Navigation subscribes to incoming links. To do so, you can provide your own `getInitialURL` and `subscribe` functions:

```
const linking = {
  prefixes: ['myapp://', 'https://myapp.com'],

  // Custom function to get the URL which was used to open the app
  async getInitialURL() {
    // First, you would need to get the initial URL from your third-party
    // integration
    // The exact usage depend on the third-party SDK you use
    // For example, to get the initial URL for Firebase Dynamic Links:
    const { isAvailable } = utils().playServicesAvailability;

    if (isAvailable) {
      const initialLink = await dynamicLinks().getInitialLink();

      if (initialLink) {
```



```
    return initialLink.url;
  }
}

// As a fallback, you may want to do the default deep link handling
const url = await Linking.getInitialURL();

return url;
},


// Custom function to subscribe to incoming links
subscribe(listener) {
  // Listen to incoming links from Firebase Dynamic Links
  const unsubscribeFirebase = dynamicLinks().onLink(({ url }) => {
    listener(url);
  });

  // Listen to incoming links from deep linking
  const linkingSubscription = Linking.addEventListener('url', ({ url }) => {
    listener(url);
  });

  return () => {
    // Clean up the event listeners
    unsubscribeFirebase();
    linkingSubscription.remove();
  };
},

config: {
  // Deep link configuration
},
};
```

Similar to the above example, you can integrate any API that provides a way to get the initial URL and to subscribe to new incoming URLs using the `getInitialURL` and `subscribe` options.

 [Edit this page](#)