Using TypeScript

TypeScript is a language which extends JavaScript by adding type definitions. New React Native projects target TypeScript by default, but also support JavaScript and Flow.

Getting Started with TypeScript

New projects created by the React Native CLI or popular templates like Ignite will use TypeScript by default.

TypeScript may also be used with Expo, which maintains TypeScript templates, or will prompt you to automatically install and configure TypeScript when a `.ts` or `.tsx` file is added to your project.

```
npx create-expo-app --template
```

Adding TypeScript to an Existing Project

1. Add TypeScript, types, and ESLint plugins to your project.

npm **Yarn**

```
yarn add --dev @tsconfig/react-native @types/jest @types/react @types/react-test-renderer
typescript
```

> **NOTE**
>
> This command adds the latest version of every dependency. The versions may need to be changed to match the existing packages used by your project. You can use a tool like React Native Upgrade Helper to see the versions shipped by React Native.

2. Add a TypeScript config file. Create a `tsconfig.json` in the root of your project:

```
{
  "extends": "@tsconfig/react-native/tsconfig.json"
}
```

3. Rename a JavaScript file to be `*.tsx`

> You should leave the `./index.js` entrypoint file as it is otherwise you may run into an issue when it comes to bundling a production build.

4. Run `yarn tsc` to type-check your new TypeScript files.

Using JavaScript Instead of TypeScript

React Native defaults new applications to TypeScript, but JavaScript may still be used. Files with a `.jsx` extension are treated as JavaScript instead of TypeScript, and will not be typechecked. JavaScript modules may still be imported by TypeScript modules, along with the reverse.

How TypeScript and React Native works

Out of the box, TypeScript sources are transformed by Babel during bundling. We recommend that you use the TypeScript compiler only for type checking. This is the default behavior of `tsc` for newly created applications. If you have existing TypeScript code being ported to React Native, there are one or two caveats to using Babel instead of TypeScript.

What does React Native + TypeScript look like

You can provide an interface for a React Component's Props and State via `React.Component<Props, State>` which will provide type-checking and editor auto-completing when working with that component in JSX.

components/Hello.tsx

```
import React from 'react';
import {Button, StyleSheet, Text, View} from 'react-native';

export type Props = {
  name: string;
  baseEnthusiasmLevel?: number;
};

const Hello: React.FC<Props> = ({
  name,
  baseEnthusiasmLevel = 0,
}) => {
  const [enthusiasmLevel, setEnthusiasmLevel] = React.useState(
    baseEnthusiasmLevel,
  );

  const onIncrement = () =>
    setEnthusiasmLevel(enthusiasmLevel + 1);
  const onDecrement = () =>
    setEnthusiasmLevel(
      enthusiasmLevel > 0 ? enthusiasmLevel - 1 : 0,
    );

  const getExclamationMarks = (numChars: number) =>
    numChars > 0 ? Array(numChars + 1).join('!') : '';

  return (
    <View style={styles.container}>
      <Text style={styles.greeting}>
        Hello {name}
        {getExclamationMarks(enthusiasmLevel)}
      </Text>
      <View>
        <Button
          title="Increase enthusiasm"
          accessibilityLabel="increment"
          onPress={onIncrement}
          color="blue"
        />
        <Button
          title="Decrease enthusiasm"
          accessibilityLabel="decrement"
          onPress={onDecrement}
          color="red"
        />
      </View>
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
  },
  greeting: {
    fontSize: 20,
    fontWeight: 'bold',
    margin: 16,
  },
});

export default Hello;
```

You can explore the syntax more in the TypeScript playground.

Where to Find Useful Advice

- TypeScript Handbook
- React's documentation on TypeScript
- React + TypeScript Cheatsheets has a good overview on how to use React with TypeScript

Using Custom Path Aliases with TypeScript

To use custom path aliases with TypeScript, you need to set the path aliases to work from both Babel and TypeScript. Here's how:

1. Edit your `tsconfig.json` to have your custom path mappings. Set anything in the root of `src` to be available with no preceding path reference, and allow any test file to be accessed by using `tests/File.tsx`:

```
{
-  "extends": "@tsconfig/react-native/tsconfig.json"
+  "extends": "@tsconfig/react-native/tsconfig.json",
+  "compilerOptions": {
+    "baseUrl": ".",
+    "paths": {
+      "*": ["src/*"],
+      "tests": ["tests/*"],
+      "@components/*": ["src/components/*"],
+    },
+  }
}
```

2. Add `babel-plugin-module-resolver` as a development package to your project:

npm **Yarn**

```
yarn add --dev babel-plugin-module-resolver
```

3. Finally, configure your `babel.config.js` (note that the syntax for your `babel.config.js` is different from your `tsconfig.json`):

```
{
   presets: ['module:metro-react-native-babel-preset'],
+  plugins: [
+    [
+       'module-resolver',
+       {
+         root: ['./src'],
+         extensions: ['.ios.js', '.android.js', '.js', '.ts', '.tsx', '.json'],
+         alias: {
+           tests: ['./tests/'],
+           "@components": "./src/components",
+         }
+       }
+    ]
+  ]
}
```

Is this page useful? 👍 👎

Edit this page

Last updated on ***Jun 21, 2023***