

Use TypeScript - Expo Documentation

 docs.expo.dev/guides/typescript/



Use TypeScript

An in-depth guide on configuring an Expo project with TypeScript.

Expo has first-class support for TypeScript. The JavaScript interface of the Expo SDK is completely written in TypeScript.



with-typescript

See the example project on GitHub.



Get started

Quick start with a template

The easiest way to get started is to initialize your new project using a TypeScript template:

 Terminal

```
- npx create-expo-app -t expo-template-blank-typescript
```

For npm, add the following **script** to the **package.json**:

📄 package.json

```
{
  "scripts": {
    "ts:check": "tsc"
    ...
  }
}
```

Then, to type-check the project, run the following command:

📄 Terminal

```
- npm run ts:check
```

When you create new source files in your project you should use the **.ts** extension or the **.tsx** if the file includes React components.

In an existing project

Rename files to convert them to TypeScript. For example, rename **App.js** to **App.tsx**. Use the **.tsx** extension if the file includes React components (JSX). If the file does not include any JSX, you can use the **.ts** file extension.

📄 Terminal

```
- mv App.js App.tsx
```



For SDK 48 and higher, running **npx expo start** prompts you to install the required dependencies, such as **typescript** and **@types/react**.

For npm, add the following **script** to the **package.json**:

package.json

```
{
  "scripts": {
    "ts:check": "tsc"
    ...
  }
}
```

You can now run `npm run ts:check` or `yarn tsc` to type-check the project.

Base configuration



You can disable the TypeScript setup in Expo CLI with the environment variable `EXPO_NO_TYPESCRIPT_SETUP=1`

A project's **tsconfig.json** should extend the `expo/tsconfig.base` by default. This sets the following default compiler options (which can be overwritten in your project's **tsconfig.json**):

You can automatically generate a **tsconfig.json** file by running the command:

Terminal

```
- npx expo customize tsconfig.json
```

Project configuration

Expo CLI will automatically modify your **tsconfig.json** to the preferred default which is optimized for universal React development:

tsconfig.json

```
{
  "extends": "expo/tsconfig.base",
  "compilerOptions": {},
  "include": ["**/*.ts", "**/*.tsx", ".expo/types/**/*.ts", "expo-env.d.ts"]
}
```

The default configuration for TypeScript is user-friendly and encourages adoption. However, if you prefer strict type checking, you can enable it by adding `"strict": true` to the `compilerOptions`. We recommend enabling this to minimize the chance of introducing runtime errors.

Some language features may require additional configuration. For example, if you want to use decorators you'll need to add the `experimentalDecorators` option. For more information on the available properties see the TypeScript compiler options documentation.

Path aliases

Expo CLI supports path aliases in your project's **tsconfig.json** automatically. This enables you to import modules using a custom alias instead of a relative path.

For example, if you have a file at **src/components/Button.tsx** and wish to import it using the alias **@/components/Button** as follows:

```
import Button from '@/components/Button';
```

Then simply add the alias **@/*** in the project's **tsconfig.json** and set it to the **src** directory:

 tsconfig.json

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@/*": ["src/*"]
    }
  }
}
```

Consider the following when using path aliases:

- Restart Expo CLI after changing **tsconfig.json** to update path aliases. You don't need to clear the Metro cache when the aliases change.
- If not using TypeScript, **jsconfig.json** can serve as an alternative to **tsconfig.json**.
- Path aliases add additional resolution time when defined.
- Path aliases are only supported by Metro (including Metro web) and not by the deprecated `@expo/webpack-config` package.
- Bare projects require additional setup for this feature. See the versioned Metro setup guide for more information.

`tsconfigPaths` is enabled by default. You can disable it by setting `tsconfigPaths` to `false` in the project's app config:

 app.json

```
{
  "expo": {
    "experiments": {
      "tsconfigPaths": false
    }
  }
}
```

Absolute imports




Available in SDK 49 and higher.

In SDK 49 projects, you'll need to enable absolute imports in the project's app config:

 app.json

```
{
  "expo": {
    "experiments": {
      "tsconfigPaths": true
    }
  }
}
```

Absolute imports from the project root directory are enabled when `compilerOptions.baseUrl` is defined in the project's **tsconfig.json** or **jsconfig.json** file. For example:

 tsconfig.json

```
{
  "compilerOptions": {
    "baseUrl": "./"
  }
}
```

Will enable the following import:

```
import Button from 'src/components/Button';
// Imports `<compilerOptions.baseUrl>/src/components/Button`
```

Consider the following when using absolute imports:

- `compilerOptions.paths` are resolved relative to the `compilerOptions.baseUrl` if it is defined, otherwise they're resolved against the project root directory.

- `compilerOptions.baseUrl` is resolved before node modules. This means if you have a file named `./path.js` in the project, it may be imported instead of a node module named `path`.
- Restarting Expo CLI is necessary to update `compilerOptions.baseUrl` after modifying the **`tsconfig.json`**.
- If you're not using TypeScript, **`jsconfig.json`** can serve as an alternative to **`tsconfig.json`**.
- Absolute imports are only supported by Metro (including Metro web) and not by `@expo/webpack-config`.
- Bare projects require additional setup for this feature. See the versioned Metro setup guide for more information.

Type generation

Some Expo libraries provide both static types and type generation capabilities. These types are automatically generated when the project builds or by running the `npm run expo customize tsconfig.json` command.

TypeScript for config files

If you want to use TypeScript for configuration files such as **`webpack.config.js`**, **`metro.config.js`**, or **`app.config.js`**, additional setup is needed. You can utilize the `ts-node` require hook to import TypeScript files within your JS config file, allowing TypeScript imports while keeping the root file as JavaScript.

Terminal

```
- npm install ts-node typescript --save-dev
```

webpack.config.js



Install the `@expo/webpack-config` package.

webpack.config.js

```
require('ts-node/register');
module.exports = require('./webpack.config.ts');
```

webpack.config.ts

```
import createExpoWebpackConfigAsync from '@expo/webpack-config/webpack';
import { Arguments, Environment } from '@expo/webpack-config/webpack/types';

module.exports = async function (env: Environment, argv: Arguments) {
  const config = await createExpoWebpackConfigAsync(env, argv);
  // Customize the config before returning it.
  return config;
};
```

metro.config.js

 metro.config.js

```
require('ts-node/register');
module.exports = require('./metro.config.ts');
```

 metro.config.ts

```
import { getDefaultConfig } from 'expo/metro-config';

const config = getDefaultConfig(__dirname);

module.exports = config;
```

app.config.js

app.config.ts is supported by default. However, it doesn't support external TypeScript modules, or **tsconfig.json** customization. You can use the following approach to get a more comprehensive TypeScript setup:

 app.config.ts

```
import 'ts-node/register'; // Add this to import TypeScript files
import { ExpoConfig } from 'expo/config';

// In SDK 46 and lower, use the following import instead:
// import { ExpoConfig } from '@expo/config-types';

const config: ExpoConfig = {
  name: 'my-app',
  slug: 'my-app',
};

export default config;
```

Learn how to use TypeScript

A good place to start learning TypeScript is the official TypeScript Handbook.

For TypeScript and React components, we recommend referring to the React TypeScript CheatSheet to learn how to type your React components in a variety of common situations.