Static Type Checking

Static type checkers like Flow and TypeScript identify certain types of problems before you even run your code. They can also improve developer workflow by adding features like auto-completion. For this reason, we recommend using Flow or TypeScript instead of `PropTypes` for larger code bases.

Flow

Flow is a static type checker for your JavaScript code. It is developed at Facebook and is often used with React. It lets you annotate the variables, functions, and React components with a special type syntax, and catch mistakes early. You can read an introduction to Flow to learn its basics.
To use Flow, you need to:

- Add Flow to your project as a dependency.
- Ensure that Flow syntax is stripped from the compiled code.
- Add type annotations and run Flow to check them.

We will explain these steps below in detail.

Adding Flow to a Project

First, navigate to your project directory in the terminal. You will need to run the following command:
If you use Yarn, run:

```
yarn add --dev flow-bin
```

If you use npm, run:

```
npm install --save-dev flow-bin
```

This command installs the latest version of Flow into your project.
Now, add `flow` to the `"scripts"` section of your `package.json` to be able to use this from the terminal:

```
{
  // ...
  "scripts": {
    "flow": "flow",
    // ...
  },
  // ...
}
```

Finally, run one of the following commands:
If you use Yarn, run:

```
yarn run flow init
```

If you use npm, run:

```
npm run flow init
```

This command will create a Flow configuration file that you will need to commit.

Stripping Flow Syntax from the Compiled Code

Flow extends the JavaScript language with a special syntax for type annotations. However, browsers aren't aware of this syntax, so we need to make sure it doesn't end up in the compiled JavaScript bundle that is sent to the browser.
The exact way to do this depends on the tools you use to compile JavaScript.

Create React App

If your project was set up using Create React App, congratulations! The Flow annotations are already being stripped by default so you don't need to do anything else in this step.

Babel

> **Note:**
> These instructions are *not* for Create React App users. Even though Create React App uses Babel under the hood, it is already configured to understand Flow. Only follow this step if you *don't* use Create React App.

If you manually configured Babel for your project, you will need to install a special preset for Flow.
If you use Yarn, run:

```
yarn add --dev babel-preset-flow
```

If you use npm, run:

```
npm install --save-dev babel-preset-flow
```

Then add the `flow` preset to your Babel configuration. For example, if you configure Babel through `.babelrc` file, it could look like this:

```
{
  "presets": [
    "flow",
    "react"
  ]
}
```

This will let you use the Flow syntax in your code.

> **Note:**
> Flow does not require the `react` preset, but they are often used together. Flow itself understands JSX syntax out of the box.

Other Build Setups

If you don't use either Create React App or Babel, you can use flow-remove-types to strip the type annotations.

Running Flow

If you followed the instructions above, you should be able to run Flow for the first time.

```
yarn flow
```

If you use npm, run:

```
npm run flow
```

You should see a message like:

```
No errors!
✨  Done in 0.17s.
```

Adding Flow Type Annotations

By default, Flow only checks the files that include this annotation:

```
// @flow
```

Typically it is placed at the top of a file. Try adding it to some files in your project and run `yarn flow` or `npm run flow` to see if Flow already found any issues.
There is also an option to force Flow to check *all* files regardless of the annotation. This can be too noisy for existing projects, but is reasonable for a new project if you want to fully type it with Flow.
Now you're all set! We recommend to check out the following resources to learn more about Flow:

- Flow Documentation: Type Annotations
- Flow Documentation: Editors
- Flow Documentation: React
- Linting in Flow

TypeScript

TypeScript is a programming language developed by Microsoft. It is a typed superset of JavaScript, and includes its own compiler. Being a typed language, TypeScript can catch errors and bugs at build time, long before your app goes live. You can learn more about using TypeScript with React here.
To use TypeScript, you need to:

- Add TypeScript as a dependency to your project
- Configure the TypeScript compiler options
- Use the right file extensions
- Add definitions for libraries you use

Let's go over these in detail.

Using TypeScript with Create React App

Create React App supports TypeScript out of the box.
To create a **new project** with TypeScript support, run:

```
npx create-react-app my-app --typescript
```

You can also add it to an **existing Create React App project**, as documented here.

> **Note:**
> If you use Create React App, you can **skip the rest of this page**. It describes the manual setup which doesn't apply to Create React App users.

Adding TypeScript to a Project

It all begins with running one command in your terminal.
If you use Yarn, run:

```
yarn add --dev typescript
```

If you use npm, run:

```
npm install --save-dev typescript
```

Congrats! You've installed the latest version of TypeScript into your project. Installing TypeScript gives us access to the `tsc` command. Before configuration, let's add `tsc` to the "scripts" section in our `package.json`:

```
{
  // ...
  "scripts": {
    "build": "tsc",
    // ...
  },
  // ...
}
```

Configuring the TypeScript Compiler

The compiler is of no help to us until we tell it what to do. In TypeScript, these rules are defined in a special file called `tsconfig.json`. To generate this file:
If you use Yarn, run:

```
yarn run tsc --init
```

If you use npm, run:

```
npx tsc --init
```

Looking at the now generated `tsconfig.json`, you can see that there are many options you can use to configure the compiler. For a detailed description of all the options, check here.
Of the many options, we'll look at `rootDir` and `outDir`. In its true fashion, the compiler will take in typescript files and generate javascript files. However we don't want to get confused with our source files and the generated output.
We'll address this in two steps:

- Firstly, let's arrange our project structure like this. We'll place all our source code in the `src` directory.

```
├── package.json
├── src
│   └── index.ts
└── tsconfig.json
```

- Next, we'll tell the compiler where our source code is and where the output should go.

```
// tsconfig.json

{
  "compilerOptions": {
    // ...
    "rootDir": "src",
    "outDir": "build"
    // ...
  },
}
```

Great! Now when we run our build script the compiler will output the generated javascript to the `build` folder. The TypeScript React Starter provides a `tsconfig.json` with a good set of rules to get you started.
Generally, you don't want to keep the generated javascript in your source control, so be sure to add the build folder to your `.gitignore`.

File extensions

In React, you most likely write your components in a `.js` file. In TypeScript we have 2 file extensions:
`.ts` is the default file extension while `.tsx` is a special extension used for files which contain `JSX`.

Running TypeScript

If you followed the instructions above, you should be able to run TypeScript for the first time.

```
yarn build
```

If you use npm, run:

```
npm run build
```

If you see no output, it means that it completed successfully.

Type Definitions

To be able to show errors and hints from other packages, the compiler relies on declaration files. A declaration file provides all the type information about a library. This enables us to use javascript libraries like those on npm in our project.
There are two main ways to get declarations for a library:
Bundled - The library bundles its own declaration file. This is great for us, since all we need to do is install the library, and we can use it right away. To check if a library has bundled types, look for an `index.d.ts` file in the project. Some libraries will have it specified in their `package.json` under the `typings` or `types` field.
DefinitelyTyped - DefinitelyTyped is a huge repository of declarations for libraries that don't bundle a declaration file. The declarations are crowd-sourced and managed by Microsoft and open source contributors. React for example doesn't bundle its own declaration file. Instead we can get it from DefinitelyTyped. To do so enter this command in your terminal.

```
# yarn
yarn add --dev @types/react

# npm
npm i --save-dev @types/react
```

Local Declarations Sometimes the package that you want to use doesn't bundle declarations nor is it available on DefinitelyTyped. In that case, we can have a local declaration file. To do this, create a `declarations.d.ts` file in the root of your source directory. A simple declaration could look like this:

```
declare module 'querystring' {
  export function stringify(val: object): string
  export function parse(val: string): object
}
```

You are now ready to code! We recommend to check out the following resources to learn more about TypeScript:

- TypeScript Documentation: Basic Types
- TypeScript Documentation: Migrating from Javascript
- TypeScript Documentation: React and Webpack

Reason

Reason is not a new language; it's a new syntax and toolchain powered by the battle-tested language, OCaml. Reason gives OCaml a familiar syntax geared toward JavaScript programmers, and caters to the existing NPM/Yarn workflow folks already know.

Reason is developed at Facebook, and is used in some of its products like Messenger. It is still somewhat experimental but it has dedicated React bindings maintained by Facebook and a vibrant community.

Kotlin

Kotlin is a statically typed language developed by JetBrains. Its target platforms include the JVM, Android, LLVM, and JavaScript.

JetBrains develops and maintains several tools specifically for the React community: React bindings as well as Create React Kotlin App. The latter helps you start building React apps with Kotlin with no build configuration.

Other Languages

Note there are other statically typed languages that compile to JavaScript and are thus React compatible. For example, F#/Fable with elmish-react. Check out their respective sites for more information, and feel free to add more statically typed languages that work with React to this page!

このページを編集する