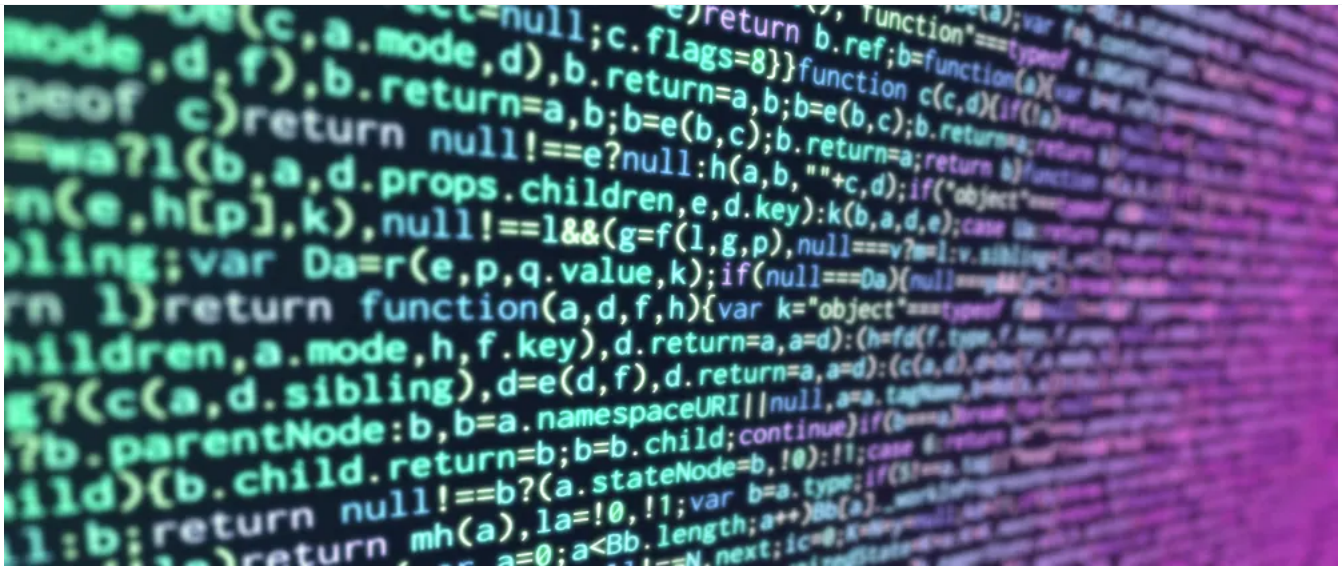

Aleksei Berezkin
Posted on Jun 14, 2021 • Updated on Sep 20

Setting up React + TypeScript + webpack app from scratch without create-react-app

#react #typescript #webpack #webdev

Artwork: https://code-art.pictures/
Code on artwork: React JS

Why bother if there is create-react-app?

Good question! In fact, if you are happy with `create-react-app` — just use it 🙂 But if you want to figure out how everything works together, let's combine all parts ourselves!

Dear reader and my fellow dev,

I try my best keeping the post up to date. However, if you find anything wrong or outdated, please write a comment, thank you!

Structure of the project we are going to create

```
/hello-react
  /dist
    index.html
    main.css
    main.js
    main.js.LICENSE.txt
  /src
    index.css
    index.tsx
  index.html
  package.json
  tsconfig.json
  webpack.config.js
```

1. Install Node.js and npm

Node.js installation steps depend on your system — just proceed to a download page and follow instructions.

npm doesn't need any installation because it comes with Node. If you wish to check that everything is properly installed on your system, follow these instructions.

> *Sidenotes: node and npm are not the only options. There's Deno which is alternative to node, and Yarn which is alternative to npm. If uncertain, I'd recommend staying with node + npm for the moment.*

2. Create the project

Create the project root dir, `hello-react`, and run `npm init` wizard from inside it:

```
mkdir hello-react
cd hello-react
npm init
```

The wizard creates an empty project asking you questions one by one. To automatically accept all default answers, append `-y` param to `npm init` command. Once wizard finishes, it creates the following file:

package.json (created by `npm init`)

```
{
  "name": "hello-react",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Not much, but... that's already a valid Node.js project!

3. Install TypeScript

Staying in the project root dir run this:

```
npm i --save-dev typescript
```

4. Create tsconfig.json

That's TypeScript configuration for the project. Create it in the project root dir and insert the following content:

tsconfig.json

```
{
  "compilerOptions": {
    "esModuleInterop": true,
    "jsx": "react",
    "module": "esnext",
    "moduleResolution": "node",
    "lib": [
      "dom",
      "esnext"
    ],
    "strict": true,
    "sourceMap": true,
    "target": "esnext",
  },
  "exclude": [
    "node_modules"
  ]
}
```

What these mean? Let's see!

- `compilerOptions`
  - `esModuleInterop` — the flag fixes default and namespace imports from CommonJS to TS. That's just needed 🙂
  - `jsx` — tells TS how to treat JSX files
  - `module` — the option tells TS how to transpile ES6 imports and exports; `esnext` leaves them unchanged. I recommend setting always `esnext` to leave this job to webpack.
  - `moduleResolution` — historically TS used to resolve modules in other way than Node.js, so this must be set to `node`
  - `lib` — this option tells TS which libraries will exist in your target environment, so TS implicitly imports their types. TS won't be able to check if these libs really exist in runtime, so that's your promise. More on this later.
  - `strict` — enables all TS type checks
  - `sourceMap` — enables TS emitting source maps. We will configure webpack to ignore source maps in production builds.
  - `target` — configures target ES version which depends on your users; more on this later.
- `exclude` — this option excludes libs from typechecking and transpiling; however your code is still checked against typedefs provided by libs.

Full `tsconfig.json` reference is here.

5. Install webpack, plugins and loaders

Staying in the project root dir, execute the following command. It's long, so make sure you scrolled enough and copied the whole line!

```
npm i --save-dev webpack webpack-cli webpack-dev-server css-loader html-webpack-plugin mini-css-extract-plugin ts-loader
```

6. Create webpack.config.js

Create `webpack.config.js` in the project root dir, and insert the following content:

webpack.config.js

```
const prod = process.env.NODE_ENV === 'production';

const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {
  mode: prod ? 'production' : 'development',
  entry: './src/index.tsx',
  output: {
    path: __dirname + '/dist/',
  },
  module: {
    rules: [
      {
        test: /\.(ts|tsx)$/,
        exclude: /node_modules/,
        resolve: {
          extensions: ['.ts', '.tsx', '.js', '.json'],
        },
        use: 'ts-loader',
      },
      {
        test: /\.css$/,
        use: [MiniCssExtractPlugin.loader, 'css-loader'],
      },
    ]
  },
  devtool: prod ? undefined : 'source-map',
  plugins: [
    new HtmlWebpackPlugin({
      template: 'index.html',
    }),
    new MiniCssExtractPlugin(),
  ],
```

```
};
```

A lot of things are going on here! webpack configuration is arguably the most complex thing in the whole setup. Let's see its parts:

- Setting a `NODE_ENV` var is the typical way of setting a dev/prod mode. See later how to set it in your script.
- `HtmlWebpackPlugin` generates `index.html` from a template which we are going to create shortly
- `MiniCssExtractPlugin` extracts styles to a separate file which otherwise remain in `index.html`
- `mode` tells webpack if your build is for development or production. In production mode webpack minifies the bundle.
- `entry` is a module to execute first after your app is loaded on a client. That's a bootstrap that will launch your application.
- `output` sets the target dir to put compiled files to
- `module.rules` describes how to load (import) different files to a bundle
  - `test: /\.(ts|tsx)$/` item loads TS files with `ts-loader`
  - `test: /\.css$/` item loads CSS files
- `devtool` sets the config for source maps
- `plugins` contains all plugins with their settings

Phew! The most complex part is behind.

7. Add scripts to package.json

Add `start` and `build` scripts to your `package.json`:

package.json

```
{
  ...
  "scripts": {
    "start": "webpack serve --port 3000",
    "build": "NODE_ENV=production webpack"
  }
  ...
}
```

These are:

- `start` launches a dev server on port 3000. Dev server automatically watches your files and rebuilds the app when needed.
- `build` builds your app for production. `NODE_ENV=production` sets `NODE_ENV` which is checked in the first line of `webpack.conf.js`. Note: On Windows PowerShell the command must be `set NODE_ENV=production && webpack`, see this.

8. Create index.html template

`HtmlWebpackPlugin` can generate HTML even without a template. However, you are likely going to need one, so let's create it in the project root dir. It's the file we referenced from `webpack.config.js` plugins section.

index.html

```
<!DOCTYPE html>
<html>
<head lang="en">
  <title>Hello React</title>
</html>
<body>
  <div id="app-root">App is loading...</div>
</body>
```

9. Install React

Staying in the project root dir, run the following:

```
npm i react react-dom
```

And then:

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

10. Create src/index.tsx

That's the entry point of your application; we've referenced it from `webpack.config.js`. You may also fix `main` to point to the same file in `package.json`, though it's not required.

src/index.tsx

```
import React from 'react'
import { createRoot } from 'react-dom/client'

const container = document.getElementById('app-root')!
const root = createRoot(container)
root.render(<h1>Hello React!</h1>)
```

Note: `createRoot()` API is new to React 18. If you need an older version, you may read this blogpost, and use the following code:

▶ src/index.tsx for React 17

11. Create src/index.css and import it to src/index.tsx

To make sure our CSS plugin works, let's apply some styles.

src/index.css

```
body {
  color: blue;
}
```

src/index.tsx

```
import './index.css'
// The rest app remains the same
// ...
```

12. Run dev server

It was very long path! But we are close to the end. Let's run the dev server:

```
npm start
```

Now open http://localhost:3000/ in your browser — you should see the colored greeting:

> ## Hello React!

Now try to modify `src/index.tsx`, for example, change a message — app must reload and show an updated text; try also change styles — they must be also picked up without server restart.

13. Build your app for production

Staying in project root dir, run this:

```
npm run build
```

You should observe appeared `dist` folder with bundle files generated. Let's try serving them as in real production:

```
npx serve dist
```

serve is a simple Node.js program that serves static files. Once launched, it outputs the URL it serves your app from, usually it's http://localhost:3000/. Open it — you should see the greeting.

14. Targeting older environments

That's a bit advanced part, so you may postpone it until you are comfortable with the basic setup.

14.1. Target ES version

Target ES is set in `tsconfig.json`: `compilerOptions.target`, and it depends on who you write your app for. So who is your user?

- You and your team — my bet you don't use anything obsolete 🙂 So it's safe to leave `esnext`
- Average Internet user — my guess would be `es<currentYear-3>`, i.e. on a year of this writing (2021) it'd be `es2018`. Why not `esnext`? There may be interesting surprises even in seemingly recent devices, for example, Xiaomi MIUI Browser 12.10.5-go released on 2021 May does not support nullish coalesce operator, here's a pen for Xiaomi users. What's your result?
- IE users — then target must be `es5`. Note: some ES6+ features get bloated when transpiled to ES5.

14.2. Select target libs

Libs are set in `tsconfig.json`: `compilerOptions.lib`, and the option also depends on your guess about your user.

Typical libs:

- `dom` — this includes all APIs provided by the browser
- `es...`, for example `es2018` — this includes JavaScripts builtins coming with corresponding ES specification.

Important: unlike Babel, these options don't add any polyfills, so if you target older environments you have to add them like described next.

14.3. Add polyfills

This depends on APIs your app needs.

- React requires: Map, Set and requestAnimationFrame which do not exist in old browsers
- If your client code uses some relatively new API like flatMap or fetch while targeting older browsers, consider polyfilling them as well.

Here are some popular polyfills:

- core-js for missing Set, Map, Array.flatMap etc
- raf for missing requestAnimationFrame
- whatwg-fetch for missing `fetch`. Note: it doesn't include `Promise` polyfill. It's included in above-mentioned `core-js`.

Given we decided to use all of them, the setup is the following:

```
npm i core-js raf whatwg-fetch
```

index.tsx

```
import 'core-js/features/array/flat-map'
import 'core-js/features/map'
import 'core-js/features/promise'
import 'core-js/features/set'
import 'raf/polyfill'
import 'whatwg-fetch'

// The rest app remains the same
// ...
```

14.4. Tweaking `webpack` runtime

It may be surprising, but even if you transpile to ES5, `webpack` may emit the code with ES6+ constructs. So if you target something ancient like IE, you may need to turn them off. See this.

Is it fair adding so much polyfills?

No, it's not given most users have quite a good browser and just wasting their runtime and bandwidth. So the best option would be making 2 bundles: for old and new environments, and load only one of them. The topic falls outside of this tutorial.

You're done!

I know that wasn't that easy 😅 But I'm sure these things are no more a puzzle for you. Thanks for staying with me for this journey!

Top comments (11)

Simon Taylor • Jun 6 '22

@alekseiberezkin awesome article, thankyou!

Martin Dwyer • Mar 6

Excellent resource! Thanks for keeping it up to date!

Yoel Duran • Feb 2

Great job. congrats

Vadim Ryabov • Aug 13 '22

12.1 - compilerOptions.ta*R*get 😉

Aleksei Berezkin • Oct 7 '22

Thx, fixed

Miguel Roche • Sep 15

@alekseiberezkin is there any chance of changing webpack for esbuild? thanks for the article

Aleksei Berezkin • Sep 20

Thx for the suggestion. You are right, esbuild is not more an “emerging” tool but a comprehensive alternative. Unfortunately I can't give any estimate when I'm able to look at it closely because right now I'm not working in frontend fulltime. However if I decide writing about something here, that'd be among the first topics to consider.

Ivan gsp • Dec 18 '21

Thanks @alekseiberezkin for the tutorial, it was really helpful but I think you missed to add `@types/react` and `@types/react-dom`

Aleksei Berezkin • Dec 20 '21

Thank you so much, updated the post

Vishal Raj • Jun 14 '21

@alekseiberezkin Thank you for the article. Will definitely try this. Meanwhile, I have a similar article, that you might like - React from scratch

Aleksei Berezkin • Jun 14 '21

Thanks for your link. Will keep it in mind if I need Babel.

Code of Conduct • Report abuse

DEV Community

Trending in Webdev

The Webdev community is currently discussing ways to build a `Meetup.com` clone using **React** and **SuperTokens**, creating syntax highlighting with **CSS** and **gradients**, and designing an animated navbar inspired by **Vercel**. There's also interest in visualizing web frameworks as superheroes and creating a full-stack **Airbnb** clone with **Next.js**, **Tailwind CSS**, **Zustand**, and **Amplication**.

🔥 2 TRICKS 🔥 to build a Meetup.com clone with React in 30 minutes □✨

Nevo David for novu · Aug 14

#webdev #javascript #react #programming

Impossible⁉️ CSS only syntax highlighting 😱 ...with a single element and GRADIENTS 🤯

GrahamTheDev · Aug 16

#css #html #javascript #webdev

Creating an animated navbar inspired by Vercel using React (Next.js v13), Framer-Motion, and Tailwind CSS

ashish · Aug 9

#webdev #tutorial #react #nextjs

🤖 We asked AI to imagine web frameworks as superheroes 🦸 💪

Matija Sosic · Aug 14

#webdev #javascript #react #gpt3

Full Stack Airbnb Clone with Next.js, Tailwind CSS, Zustand and Amplication

Kishan Sheth · Aug 14

#nextjs #webdev #react #javascript

Aleksei Berezkin

Fullstack dev: Java, Kotlin, JS, TS, React

LOCATION
München, Deutschland

WORK
Fullstack developer

JOINED
Aug 22, 2020

More from Aleksei Berezkin

Show DEV: Interactive Mandelbrot Set Visualization
#webdev #webassembly #showdev

3 Reasons Your Site May Be Not GDPR-Compliant
#webdev #gdpr #law #eu

Code Art — abstract artworks for your creations
#showdev #webdev #webgl2 #art