Credit : https://jscomplete.com/learn/1rd-reactful

The always-recent guide to creating a development environment for Node and React (with Babel and Webpack)

This is a step-by-step guide with explanations about all the tools used. At the end of the guide, there is "sample" server-render-ready application that you can use to test your configurations.

|  |  |
| --- | --- |
|  | This guide uses tools that are popular in the JavaScript ecosystem. **Express** (for a web server), **Webpack** (for a module bundler), and **Babel** (for a JS/JSX compiler). If you want to use **TypeScript** instead of Babel, checkout [jscomplete.com/learn/reactful.ts](https://jscomplete.com/learn/reactful.ts) |

After going through these instructions, you will have a simple but fully-configured environment that’s ready for rendering React applications both on the front-end and the back-end (for server-side rendering). The instructions start with an empty directory and they will guide you through all the dependencies that you need, one-by-one. There will be no use of any "code-generation" tools like create-react-app.

## **1. Initializing**

Create an empty directory and initialize it with a package.json file. This file is used in Nodejs projects to store general information about the project (like its name, version, etc) and track what dependencies the project needs (so that anyone can install them all at once).

You can create this file using the npm init command:

$ mkdir reactful

$ cd reactful

$ npm init

The npm init command will ask you a few questions and you can interactively supply your answers (or press Enter to keep the defaults).

|  |  |
| --- | --- |
|  | You can use npm init -y to generate your package.json file with the default values that npm can detect about your project (the y is for yes to all questions). |

Once the npm init command is done, you should have a package.json file under your project directory (and nothing else, yet).

The package.json file can now be used to "document" any dependencies you need to add to your project. This happens automatically when you npm install anything.

## **2. Installing main dependencies**

A JavaScript environment has 2 types of dependencies, main dependencies that need to be installed on all environments, and development dependencies that are only needed on local development machines.

For a Nodejs web server, one popular option to use is **Express**. You can use it to serve dynamic content under your web server. (You can also use Express to serve static content, but you should consider using a better option for that, like [*NGINX*](https://www.nginx.com/) or a CDN service.)

To add Express to the project:

$ npm i express

This command will download the express npm package and place it under a node\_modules folder (which it will create because express is the first package we’re adding). The command will also save this dependency to your package.json file.

|  |  |
| --- | --- |
|  | The i in the command above is just a shortcut for install. |

The frontend dependencies you need are **React** and **ReactDOM**. Add them next:

$ npm i react react-dom

|  |  |
| --- | --- |
|  | While the react and react-dom packages are not really needed in production because they get bundled into a single file, this guide assumes that you deploy your unbundled code to production and bundle things there. If you want to bundle things in development and push your bundled files to production, you can install these packages - and most of what’s coming next - as development dependencies. |

Since you’ll be writing your code in multiple modules (files) and it will depend on other modules (like React), you need a module bundler to translate all these modules into something that can work in all browsers today. You can use **Webpack** for that job. The packages you need to install now are:

$ npm i webpack webpack-cli

|  |  |
| --- | --- |
|  | The webpack-cli package provides the webpack command, which you can use to bundle your modules. The actual Webpack core code is hosted separately under the webpack packages. |

Webpack is just a generic module bundler. You need to configure it with loaders to transform code from one state into the other. For example, you need to transform React’s JSX code into React’s API calls. The tool for that job is **Babel**. Besides JSX, Babel can also transform modern JavaScript features into code that can be understood in any execution environment. You need to install some presets as well to make all Babel transformations happen.

Here are the 5 packages that you need to make Babel do its magic:

$ npm i babel-loader @babel/core @babel/node @babel/preset-env @babel/preset-react

The babel-loader package provides the Webpack loader (which we’ll need to configure). The other @babel-scoped libraries are needed to run the Babel configurations for Nodejs and Reactjs.

## **3. Installing development dependencies**

The following are dependencies that are not needed in production. To track them separately, you can use the npm -D install flag to save them under a devDependencies section in package.json.

When you run a Node server and then change the code of that server, you need to restart Node. This will be a frustrating thing in development. Luckily, there are some workarounds. The most popular one is **Nodemon**:

$ npm i -D nodemon

This package will make the nodemon command available in your project. Nodemon runs your Node server in a wrapper process that **mon**itors the main process and automatically restarts it when files are saved to the disk. Simple and powerful!

Another priceless development dependency is [**ESLint**](https://eslint.org/). DO NOT SKIP THIS ONE!

ESLint is a code quality tool and if you don’t use it, your code will not be as good as it could be.

Since Babel is part of this stack, you need to configure ESLint to parse through what Babel is going to parse through. You should also use the main recommended ESLint configurations in addition to those recommended for React projects. Here are the packages you need for that:

$ npm i -D eslint @babel/eslint-parser eslint-plugin-react eslint-plugin-react-hooks

To configure ESLint, you need to add a .eslintrc.js file in the root of the project. This file will naturally depend on your code style preferences, but definitely start it with the recommended configurations and then customize them as needed:

.eslintrc.js

module.exports = {

parser: "@babel/eslint-parser",

env: {

browser: true,

commonjs: true,

es6: true,

node: true,

jest: true,

},

parserOptions: {

ecmaVersion: 2020,

ecmaFeatures: {

impliedStrict: true,

jsx: true,

},

sourceType: "module",

},

plugins: ["react", "react-hooks"],

extends: [

"eslint:recommended",

"plugin:react/recommended",

"plugin:react-hooks/recommended",

],

settings: {

react: {

version: "detect",

},

},

rules: {

"react/prop-types": "off",

"react/react-in-jsx-scope": "off",

// You can do more rule customizations here...

},

};

|  |  |
| --- | --- |
|  | You should configure your editor to highlight any ESLint issues for you as you type. All the major editors today have plugins to do that. You should also make your editor auto-format code for you on save. [**Prettier**](https://prettier.io/) is a great option for that and it works well with ESLint. |

The most popular testing library that’s usually used with React is [**Jest**](https://jestjs.io/). You’ll also need babel-jest and a test renderer like react-test-renderer:

$ npm i -D jest babel-jest react-test-renderer

## **4. Creating an initial directory structure**

This really depends on your style and preferences, but a simple structure would be something like:

reactful/

dist/

main.js

src/

index.js

components/

app.js

server/

server.js

|  |  |
| --- | --- |
|  | Note how I created a separate server directory for the backend code. It’s always a good idea to separate code that you run in your trusted private backends from code that is to be run on public clients. |

## **5. Configuring Webpack and Babel**

To configure Babel to compile JSX and modern JavaScript code, create a babel.config.js file under the root of the project and put the following module.exports object in it:

babel.config.js

module.exports = {

presets: [

"@babel/preset-env",

["@babel/preset-react", { "runtime": "automatic" }]

],

};

These presets were already installed above. The env preset is the one Babel uses to transform modern JavaScript (and it’s configurable if you need to target just modern browsers and make the bundle smaller). The react preset is for Babel to transform JSX into regular React API calls that browsers can understand.

To configure Webpack to bundle your application into a single bundle file, create a webpack.config.js file under the root of the project and put the following module.exports object in it:

webpack.config.js

module.exports = {

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader",

},

},

],

},

};

|  |  |
| --- | --- |
|  | Webpack has certain defaults on which JavaScript file to start with. It looks for a src/index.js file. It’ll also output the bundle to dist/main.js by default. If you need to change the locations of your src and dist files, you’ll need a few more configuration entries in webpack.config.js. |

## **6. Creating npm scripts for development**

You need 2 commands to run this environment. You need to run your web server and you need to run Webpack to bundle the frontend application for browsers. You can use **npm scripts** to manage these.

|  |  |
| --- | --- |
|  | You can also use the Webpack Dev Server instead of manually running the webpack command, but while that works great in development it’s not a good idea in production. I like my development environment to be as close as possible to the production environment. |

In your package.json file, there is a scripts section. If you generated the file with the npm init defaults you’ll have a placeholder "test" script in there. Change that to work with Jest:

// In package.json

scripts: {

"test": "jest"

}

Now, add 2 more scripts in there. The first script is to run the server file with Nodemon and make it work with the same Babel configuration above. You can name the script anything. For example:

"dev:server": "nodemon --exec babel-node src/server/server.js --ignore dist/"

|  |  |
| --- | --- |
|  | It’s probably a good idea to ignore the dist/ directory when restarting Node automatically as changes in the dist/ directory are driven by changes in the src/ directory, which is already monitored. |

The other script that you need is a simple runner for Webpack:

"dev:bundler": "webpack -w --mode=development"

The -w flag in the command above is to run Webpack in watch mode as well and the --mode=development flag is to make Webpack generate a development-friendly bundle.

|  |  |
| --- | --- |
|  | Run Webpack with --mode=production in production. |

Here’s the whole scripts section in package.json after adding these 2 new dev tasks (note the commas):

// In package.json

scripts: {

"test": "jest",

"dev:server": "nodemon --exec babel-node src/server/server.js --ignore dist/",

"dev:bundler": "webpack -w --mode=development"

}

## **7. Testing everything with a sample application**

At this point, you are ready for your own code. If you followed the exact configurations above, you’ll need to place your ReactDOM.render call (or .hydrateRoot for SSR code) in src/index.js and serve dist/main.js in your root HTML response.

Here is a sample server-side ready React application that you can test with:

src/components/app.js

import { useState } from "react";

export default function App() {

const [count, setCount] = useState(0);

return (

<div>

This is a sample stateful and server-side

rendered React application.

<br />

<br />

Here is a button that will track

how many times you click it:

<br />

<br />

<button onClick={() => setCount(count + 1)}>{count}</button>

</div>

);

}

src/index.js

import ReactDOM from "react-dom/client";

import App from "./components/app";

const container = document.getElementById("app");

ReactDOM.hydrateRoot(container, <App />);

src/server/server.js

import express from "express";

import ReactDOMServer from "react-dom/server";

import App from "../components/app";

const server = express();

server.use(express.static("dist"));

server.get("/", (req, res) => {

const initialMarkup = ReactDOMServer.renderToString(<App />);

res.send(`

<html>

<head>

<title>Sample React App</title>

</head>

<body>

<div id="app">${initialMarkup}</div>

<script src="/main.js"></script>

</body>

</html>

`)

});

server.listen(4242, () => console.log("Server is running..."));

That’s it. If you run both npm dev:server and dev:bundler scripts (in 2 separate terminals):

$ npm run dev:server

$ npm run dev:bundler

Then open up your browser on <http://localhost:4242/>, you should see the React application rendered. This application should also be rendered if you disable JavaScript in your browser!