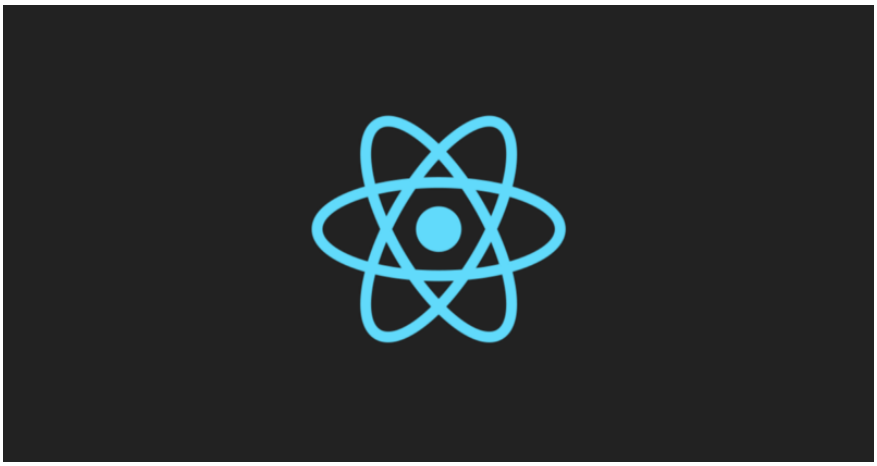
Sandeep Raveesh Follow
Apr 16 · 5 min read

Full Stack Web Application using React, Node.js, Express and Webpack



Create React App is a quick way to get started with React development and it requires no build configuration. But it completely hides the build config which makes it difficult to extend. It also requires some additional work to integrate it with an existing Node.js/Express backend application.

In this guide, we will walk through the set up of a simple full stack React application with a Node.js and Express backend. Client side code is written in React and the backend API is written using Express. This application is configured with Airbnb's ESLint rules and formatted through prettier.

Source code for this application can be found here.

Quick Start

```
# Clone the repository
git clone https://github.com/crsandeep/simple-react-full-
stack
```

```
# Go inside the directory
cd simple-react-full-stack

# Install dependencies
yarn (or npm install)

# Start development server
yarn dev (or npm run dev)

# Build for production
yarn build (or npm run build)

# Start production server
yarn start (or npm start)
```

Development mode

In the development mode, we will have 2 servers running. The front end code will be served by the webpack dev server which helps with hot and live reloading. The server side Express code will be served by a node server using nodemon which helps in automatically restarting the server whenever server side code changes.

Production mode

In the production mode, we will have only 1 server running. All the client side code will be bundled into static files using webpack and it will be served by the Node.js/Express application.

Details

Folder Structure

All the source code will be inside **src** directory. Inside src, there is client and server directory. All the frontend code (react, css, js and any other assets) will be in client directory. Backend Node.js/Express code will be in the server directory.

Babel

Babel helps us to write code in the latest version of JavaScript. If an environment does not support certain features natively, Babel will help us to compile those features down to a supported version. It also helps us to convert JSX to Javascript.

.babelrc file is used describe the configurations required for Babel. Below is the .babelrc file which I am using.

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

Babel requires plugins to do the transformation. Presets are the set of plugins defined by Babel. Preset **env** allows to use babel-preset-es2015, babel-preset-es2016, and babel-preset-es2017 and it will transform them to ES5. Preset **react** allows us to use JSX syntax and it will transform JSX to Javascript.

ESLint

ESLint is a pluggable and configurable linter tool for identifying and reporting on patterns in JavaScript.

.eslintrc.json file (alternatively configurations can we wriiten in Javascript or YAML as well) is used describe the configurations required for ESLint. Below is the .eslintrc.json file which I am using.

```
{
  "extends": ["airbnb"],
  "env": {
    "browser": true,
    "node": true
  },
  "rules": {
    "no-console": "off",
    "comma-dangle": "off",
    "react/jsx-filename-extension": "off"
  }
}
```

I am using Airbnb's Javascript Style Guide which is used by many JavaScript developers worldwide. Since we are going to write both client (browser) and server side (Node.js) code, I am setting the **env** to browser and node. Optionally, we can override the Airbnb's configurations to suit our needs. I have turned off **no-console**, **comma-dangle** and **react/jsx-filename-extension** rules.

Webpack

Webpack is a module bundler. Its main purpose is to bundle JavaScript files for usage in a browser.

webpack.config.js file is used to describe the configurations required for webpack. Below is the webpack.config.js file which I am using.

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const CleanWebpackPlugin = require("clean-webpack-plugin");


const outputDirectory = "dist";


module.exports = {
  entry: "./src/client/index.js",
  output: {
    path: path.join(__dirname, outputDirectory),
    filename: "bundle.js"
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: "babel-loader"
        }
      },
      {
        test: /\.css$/,
        use: ["style-loader", "css-loader"]
      }
    ]
  },
  devServer: {
    port: 3000,
    open: true,
    proxy: {
      "/api": "http://localhost:8080"
    }
  },
  plugins: [
    new CleanWebpackPlugin([outputDirectory]),
    new HtmlWebpackPlugin({
      template: "./public/index.html",
      favicon: "./public/favicon.ico"
    })
  ]
};
```

1. **entry:** Here the application starts executing and webpack starts bundling
2. **output path and filename:** the target directory and the filename for the bundled output
3. **module loaders:** Module loaders are transformations that are applied on the source code of a module. We pass all the js file through babel-loader to transform JSX to Javascript. CSS files are passed through css-loaders and style-loaders to load and bundle CSS files.
4. **Dev Server:** Configurations for the webpack-dev-server which will be described in coming section.
5. **plugins:** clean-webpack-plugin is a webpack plugin to remove the build folder(s) before building. html-webpack-plugin simplifies creation of HTML files to serve your webpack bundles. It loads the template (public/index.html) and injects the output bundle.

Webpack dev server

Webpack dev server is used along with webpack. It provides a development server that provides live reloading for the client side code. This should be used for development only.

The devServer section of webpack.config.js contains the configuration required to run webpack-dev-server which is given below.

```
devServer: {
    port: 3000,
    open: true,
    proxy: {
        "/api": "http://localhost:8080"
    }
}
```

Port specifies the Webpack dev server to listen on this particular port (3000 in this case). When **open** is set to true, it will automatically open the home page on startup. Proxying URLs can be useful when we have a separate API backend development server and we want to send API requests on the same domain. In our case, we have a Node.js/Express backend where we want to send the API requests to.

Nodemon

Nodemon is a utility that will monitor for any changes in the server source code and it automatically restart the server. This is used in development only.

nodemon.json file is used to describe the configurations for Nodemon. Below is the nodemon.json file which I am using.

```
{
  "watch": ["src/server/"]
}
```

Here, we tell nodemon to watch the files in the directory src/server where out server side code resides. Nodemon will restart the node server whenever a file under src/server directory is modified.

Express

Express is a web application framework for Node.js. It is used to build our backend API's.

src/server/index.js is the entry point to the server application. Below is the src/server/index.js file

```
const express = require("express");
const os = require("os");

const app = express();

app.use(express.static("dist"));
app.get("/api/getUsername", (req, res) =>
  res.send({ username: os.userInfo().username })
);
app.listen(8080, () => console.log("Listening on port
8080!"));
```

This starts a server and listens on port 8080 for connections. The app responds with `{username: <username>}` for requests to the URL (/api/getUsername). It is also configured to serve the static files from **dist** directory.

Concurrently

Concurrently is used to run multiple commands concurrently. I am using it to run the webpack dev server and the backend node server concurrently in the development environment. Below are the npm/yarn script commands used.

```
"client": "webpack-dev-server --mode development --devtool
inline-source-map --hot",
"server": "nodemon src/server/index.js",
"dev": "concurrently \"npm run server\" \"npm run client\""
```

VSCode + ESLint + Prettier

VSCode is a lightweight but powerful source code editor. ESLint takes care of the code-quality. Prettier takes care of all the formatting.

Installation guide

1. Install VSCode
2. Install ESLint extension
3. Install Prettier extension
4. Modify the VSCode user settings to add below configuration

```
"eslint.alwaysShowStatus": true,

"eslint.autoFixOnSave": true,

"editor.formatOnSave": true,

"prettier.eslintIntegration": true
```

Above, we have modified editor configurations. Alternatively, this can be configured at the project level by following this article.

Source code for this application can be found here.