Gasim Gasimzada Follow
Software Developer
May 9, 2017 · 8 min read

What are NPM, Yarn, Babel, and Webpack; and how to properly use them?

Most of us who got into building interactive web apps, started from building normal websites using libraries such as jQuery. As we move forward to starting our adventure, we first encounter these four technologies. Setting up a React project becomes a painful experience.

In this article, we will try to alleviate this painful experience by explaining these technologies one by one and how they work together.

NPM and Yarn

These two technologies solve the exact same problem. To be more concrete, Yarn is a superset of NPM that solves many problems that NPM has. So what are these for?

NPM stands for Node Package Manager. It is what its name describes. It is a package manager for Node based environments. It keeps track of all the packages and their versions and allows the developer to easily update or remove these dependencies. All of these external dependencies are being stored inside a file called called `package.json`. The initial file can be created easily using CLI `npm init` (assuming NodeJS is installed in the system). When you install a package using NPM, the packages get downloaded from a dedicated registry. There are lot of features of NPM like publishing. If you like to learn more about NPM, check out the links at the bottom.

Every language that we use has some form of package manager, either official or a 3rd party one. PHP has `Composer`, Python has `PyPi`, Java has `Gradle` etc.

Now, let’s talk briefly about Yarn. It is a package manager that uses NPM registry as its backend. Yarn has two main advantages over NPM. Firstly, Yarn creates a `yarn.lock` file. This file stores the exact versions of dependencies to the last digit. When installing, Yarn first checks the lock file for the versions, then checks `package.json` file. NPM has a `shrinkwrap` command that does exactly this. However, Yarn creates and updates its lock file automatically when dependencies are being installed/updated. Secondly, Yarn is very fast. When installing dependencies for a project, NPM installs packages sequentially. This slows down the performance significantly. Yarn solves this problem by installing these packages in parallel.

This is it for now for NPM and Yarn.

Babel

As any language, Javascript also has versions named ECMAScript (short for ES). Currently, most browsers support ES5. ES5 used to be good even though it was painful to code in it. Remember, `this` not reading from inside callback functions? The new version of Javascript, ES6, also known as ES2015 (specs of the language were finalized in June 2015) makes Javascript great again. If you want to learn about ES6, check out the links at the end of this article. All the great features of ES6 come with one big problem—majority of browsers do not fully support them. That's when Babel comes to play. Babel is a JS transpiler that converts new JS code into old ones. It is a very flexible tool in terms of transpiling. One can easily add presets such as `es2015`, `es2016`, `es2017`, so that Babel compiles them to ES5.

Here is an example—a code written in ES6:

```
class Test {
  calculatePowers() {
    return [1,2,3].map(n => n ** 2);
  }
}
```

Babel will transpile this code to the following, given the presets `es2015`:

```
"use strict";
```

```
var _createClass = function () { function
defineProperties(target, props) { for (var i = 0; i <
props.length; i++) { var descriptor = props[i];
descriptor.enumerable = descriptor.enumerable || false;
descriptor.configurable = true; if ("value" in descriptor)
descriptor.writable = true; Object.defineProperty(target,
descriptor.key, descriptor); } } return function
(Constructor, protoProps, staticProps) { if (protoProps)
defineProperties(Constructor.prototype, protoProps); if
(staticProps) defineProperties(Constructor, staticProps);
return Constructor; }; }();


function _classCallCheck(instance, Constructor) { if (!
(instance instanceof Constructor)) { throw new
TypeError("Cannot call a class as a function"); } }


var Test = function () {
  function Test() {
    _classCallCheck(this, Test);
  }


_createClass(Test, [{
    key: "calculatePowers",
    value: function calculatePowers() {
      return [1, 2, 3].map(function (n) {
        return Math.pow(n, 2);
      });
    }
  }]);


return Test;
}();
```

This is an example of how Babel allows us to have a clean, maintainable code using the latest JS specifications without needing to worry about browser support.

Webpack

Now that we know what Babel and ES6/7 are, we would like to use that. We would also like to use SASS for our styles, PostCSS for autoprefixing. Plus, we would like to minify and uglify both our CSS and Javascript code. Webpack solves all of these problems using one config file (named `webpack.config.js`) and one CLI command `webpack`.

Webpack is a modular build tool that has two sets of functionality—Loaders and Plugins. Loaders transform the source code of a module.

For example, `style-loader` adds CSS to DOM using `style` tags. `sass-loader` compiles SASS files to CSS. `babel-loader` transpiles JS code given the presets. Plugins are the core of Webpack. They can do things that loaders can't. For example, there is a plugin called `UglifyJS` that minifies and uglifies the output of webpack.

Putting them all together

Now we know concepts behind what these are, let's build a simple Hello World app using Babel, SASS, Webpack, Yarn, and React. This app will just show Hello World from inside a React component. I did not talk about React but the following toolchain is used a lot in React apps; therefore, I have chosen to show this example using React.

Firstly, let's install yarn globally. If you are on a linux system and have NodeJS installed, type in

```
sudo npm install -g yarn
```

If you are using macOS and have NodeJS and Homebrew installed, type in

```
brew install yarn
```

Now that we have Yarn installed, let's go to our working directory. Once, we are in our working directory (mine is ~/example-react-app), type in `yarn init --yes` . If you check your directory, you will now see that `package.json` file is created.

Time to install all the dependencies. We need to decide what we need for our project— **Webpack, Babel, Babel JSX syntax, Babel ES2015, SASS,** and all the necessary loaders for webpack:

```
yarn add --dev webpack babel-core babel-loader babel-preset-
react babel-preset-es2015 node-sass css-loader sass-loader
style-loader
```

We also need React:

```
yarn add react react-dom
```

You might ask why we added `--dev` flag for webpack dependencies. After installing both of these, if you check package.json file, you will see that the ones installed with `--dev` flag are in `devDependencies` array while the ones without are in `dependencies` array. If you install this project in production environment, only packages inside `dependencies` array will be installed.

. . .

Once the dependencies are installed, let's create our sample app. The app's structure will be like the following:

```
src/
   /index.jsx
   /index.html
   /components
       /HelloWorld.jsx
   /styles
       /app.scss
       /components
           /hello-world.scss
```

In src/components/HelloWorld.jsx, add the following code:

```
import React, { Component } from 'react';


export class HelloWorld extends Component {


    render() {
        return (
            <div className="hello-world">
                <h1>Hello World</h1>
            </div>
        );
    }

}

export default HelloWorld;
```

In src/index.jsx, we will include HelloWorld component and render it. We will also include our sass style:

```
import React from 'react';
import ReactDOM from 'react-dom';

import HelloWorld from './components/HelloWorld';

import './styles/app.scss';

ReactDOM.render(
   <HelloWorld />,
   document.getElementById('app')
);
```

In src/styles/app.scss, we will include the components/hello-world.scss:

```
@import 'components/hello-world';
```

In src/styles/components/hello-world.scss, change color of hello world container to red:

```
.hello-world {
    color: red;
}
```

src/index.html will load the bundle file:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Example React App</title>
</head>
<body>
   <div id="app"></div>
   <script src="/output/bundle.js"></script>
</body>
</html>
```

. . .

Now that we have added all the app structure, we need to setup Webpack. First, we create webpack.config.js in the root of working directory. Webpack needs an entry point and an output directory. Our entry point is the src/index.jsx file. Let's set our output directory to be `output` and output filename `bundle.js` . We also need to make that files that start with `.jsx` do not need the extension to be included when it is being imported.

In the webpack configuration file, let's add these two important fields:

```
// This library allows us to combine paths easily
const path = require('path');


module.exports = {
   entry: path.resolve(__dirname, 'src', 'index.jsx'),
   output: {
      path: path.resolve(__dirname, 'output'),
      filename: 'bundle.js'
   },
   resolve: {
      extensions: ['.js', '.jsx']
   }
};
```

Now, we need to set up our loaders. For files that end with extension `.jsx` , we will use **babel-loader with es2015 and react presets**. For files that end with extension `.scss` , we will use **sass-loader, css-loader, and style-loaders.** This is how our webpack.config.js will look like once we add these loaders:

```
// This library allows us to combine paths easily
const path = require('path');


module.exports = {
   entry: path.resolve(__dirname, 'src', 'index.jsx'),
   output: {
      path: path.resolve(__dirname, 'output'),
      filename: 'bundle.js'
   },
   resolve: {
      extensions: ['.js', '.jsx']
   },
   module: {
      rules: [
         {
             test: /\.jsx/,
             use: {
                loader: 'babel-loader',
                options: { presets: ['react', 'es2015'] }
             }
         },
         {
            test: /\.scss/,
            use: ['style-loader', 'css-loader', 'sass-
loader']
         }
      ]
   }
};
```

Now we need to run webpack. The easiest way to do it is to add it into `package.json`. Add the following to the JSON file's root:

```
scripts: {
    "build": "webpack -p"
}
```

The flag `-p` stands for production, which minifies and uglifies the code without needing to include the plugins in the configuration. Once you save the file, open terminal and type in `yarn run build`. If you check the output directory now, you will see bundle.js file created inside. I will not talk how to load the created bundle file as it is out of the scope of this article.

. . .

Webpack is now setup and the bundle file is being created. Time to make development easier. We need hot loading for faster development. The easiest and the best solution is to use Webpack Dev Server. First we need to update our webpack configuration. Add the following to the root of webpack config:

```
devServer: {
   contentBase: './src',
   publicPath: '/output'
}
```

These two lines tell the webpack dev server to read content from `src` directory to serve the assets under `/output` URL path. Now, we need to go to terminal and install webpack dev server

```
yarn add --dev webpack-dev-server
```

Once webpack dev server is installed, let's add another script to package.json to run it. Under scripts, add the following:

```
"dev": "webpack-dev-server --hot --inline"
```

`--hot` flag stands for hot loading, `--inline` flag stands for not showing webpack dev server toolbar. I like it that way, so I set it up that way.

Now let's run it:

```
yarn run dev
```

Webpack Dev Server runs in port 8080 by default. Open your browser and type in http://localhost:8080 and you are all set.

The example project that we created is located at the following URL: https://github.com/appristas/example-react-project

Conclusion

The purpose of this article was to explain why these tools are necessary for fastly building interactive web apps using React (or any other framework for that matter) and how these tools work together to provide fast development and one click deployment.

It is important for me to mention that Facebook has created a great tool for kickstarting react apps without needing build configuration: create-react-app. Under the hood, this starter pack also uses webpack and babel.

Links

NPM: https://www.npmjs.com
Yarn: https://yarnpkg.com
Babel: https://babeljs.io
Webpack: https://webpack.js.org

Revisions

- [10/03/2017] Added `resolve` and `extensions` to our main webpack configuration, which is necessary to import JS files without typing extensions.
- [05/11/2017] Fixed some grammar errors. Minor changes to structure.