Open in app ↗

Get unlimited access to all of Medium. Become a member

A Template for Creating a Full Stack Web Application with Flask, NPM, Webpack, and Reactjs

eyong kevin · Follow
Published in ITNEXT
18 min read · Oct 8, 2018

Listen Share More

Why are some people more successful than others? Why do some people make more money, live happier lives, and accomplish much more in the same number of years than the great majority? What is the real "secret of success?"

Unless you become more of a doer than a thinker, you would find it hard to accomplish anything meaningful. The ability to create a full-stack web application will give you the freedom to quickly bring your great idea to existence and to the reach of everybody as we all know our world is internet driven.

The aim of this article is to help you start your own full-stack web application with a Python Flask server and a React front-end. This article is suitable for anyone with a basic understanding of programming and technology who wants to explore the beauty of python-flask, npm, webpack and reactjs to build a full-stack web app.

This blog will go through the implementation which can be found in this GitHub link here. However, you will gain more if you follow along this blog post.

Folders and Files Organisation

We will start by organizing files and folders for our project. Organizing static files such as JavaScript, stylesheets, images, and so on efficiently is always a matter of concern for all web framework. Flask recommends a specific way to organize static files in our application

```
.
├── hello_template
    ├── configurations.py
    ├── __init__.py
    ├── README.md
    ├── run.py
    └── templates
        ├── hello
        │   ├── __init__.py
        │   └── views.py
        ├── __init__.py
        ├── public
        │   ├── css
        │   ├── fonts
        │   ├── images
        │   └── js
        └── static
            ├── index.html
            ├── __init__.py
            ├── js
               ├── components
               ├── index.jsx
               └── routes.js
```

We have arranged our application in a modular manner. We achieved this by using the `__init__.py` file in our folders, which are to be used as modules.

Flask Configuration

Flask Python

> *Flask is a microframework for Python based on Werkzeug, Jinja2, and good intentions*

Why micro? It simply refers to the fact that Flask aims at keeping the core of the framework small but highly extensible. This makes writing applications or extensions very easy and flexible and gives developers the power to choose the configurations they want for their application, without imposing any restrictions on the choice of database, templating engine, and so on.

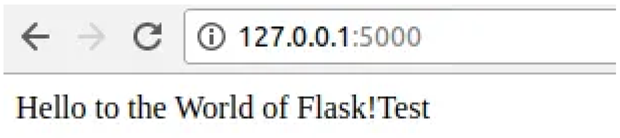
I assume here that you have already set up your Python environment. If not, following this link to install python and get things ready. Lets continue by installing Flask by running the following command `pip install Flask`

To be sure everything is working fine, let's create a fast and simple Flask-based web application. Copy the code below and paste it in the `hello_template/run.py` file and save.

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
 return 'Hello to the World of Flask!'
if __name__ == '__main__':
 app.run()
```

> *In this code as well as in any other code of this article, make sure you indent the code where appropriate*

Open a terminal in root directory `hello_template/` and execute the following command to run the application:
`python run.py`
If all is working correctly, you will be given an address `http://127.0.0.1:5000/` which you can open in your favorite browser and see our application running.



Hello to the World of Flask!Test

Our Flask app running in Chrome

> *To see all packages which Flask depends on, you may install Flask with the command* `pip install -U flask`*. The* `-U` *refers to the installation with upgrades. Also, this will list all the packages on which Flask depends, which happens to be* `flask, Werkzeug, Jinja2,` `itsdangerous`*, and* `markupsafe`*. Flask won't work if any of them are missing*

Class-based Settings: We should bare in mind that this is just a template, however, as the project gets bigger, you can have different deployment modes such as production, testing, staging, and so on, where each mode can have several different configuration settings and some settings will remain the same.
For this project, we will use a class-based setting, where we will have a default setting base class, and other classes can inherit this base class and override or add deployment-specific configuration variables.

Open the `hello_template/configurations.py` file and paste the below code.

```
class BaseConfig(object):
 '''
 Base config class
 '''
 DEBUG = True
 TESTING = False

class ProductionConfig(BaseConfig):
 """
 Production specific config
 """
 DEBUG = False
```

```
class DevelopmentConfig(BaseConfig):
 """
 Development environment specific configuration
 """
 DEBUG = True
 TESTING = True
```

Now, we can use any of the preceding classes while loading the application's configuration via `from_object()` in `hello_template/run.py` as:

```
from templates import app
#Load this config object for development mode
app.config.from_object('configurations.DevelopmentConfig')
app.run()
```

> *Enabling the debug mode will make the server reload itself in the case of any code changes, and it also provides the very helpful Werkzeug debugger when something goes wrong.*
>
> *Also, notice here that we imported `app` from templates. We are going to define our `app object` in the `__init__.py` file in the `hello_template/templates/` folder. The importation here is possible because we have arranged our application in a modular manner*

Loading Static files: If there exists a folder named `static` at the application's root level, that is, at the same level as `run.py`, then Flask will automatically read the contents of the folder without any extra configuration. Alternatively, we can provide a parameter named `static_folder` to the application object while defining the application in `hello_template/templates/__init__.py` file

```
from flask import Flask

app = Flask(__name__,
 static_folder = './public',
 template_folder="./static")

import templates.hello.views
```

Then, we will have an empty `hello_template/templates/hello/__init__.py` file just to make the enclosing folder a Python package. Finally, the `hello_template/templates/hello/views.py` will be:

```
from templates import app
from flask import render_template

@app.route('/')
@app.route('/hello')
def index():
 return render_template("index.html")
```

> *We can identify a circular import between* `hello_template/templates/__init__.py` *and* `hello_template/templates/hello/views.py`*, where, in the former, we import views from the later, and in the latter, we import the app from the former. So, this actually makes the two modules depend on each other.*

In our hello folder, we used the views to render an HTML file which will be displayed whenever it sees at the end of the url `/` and `/hello` . So the normal execution process will be:

- Start the application from `hello_template/run.py`
- `hello_template/run.py` imports app from `hello_template/templates/__init__.py`
- `hello_template/templates/__init__.py` initializes the app and imports all the views
- `hello_template/templates/hello/views.py` listen to the url `/` and `/hello` and renders an html file.

we used `render_template("index.html")` to render an HTML file instead of generating the HTML from the Python which is no fun, and actually pretty cumbersome because you have to do HTML escaping on your own to keep the application secure. And we will notice that we had configured our app with `template_folder="./static",` hence the static folder will contain our HTML template files which could be loaded directly without specifying the path again. You can read more from here

> `hello_template/templates/hello/views.py` *is React specific. We will cover how to install React later*

Making our web app modular with blueprints: A blueprint is a concept in Flask that helps make larger applications really modular. It is actually a set of operations that can be registered on an application and represents how to construct or build an application. We will modify our views to work using blueprints.

First, our `hello_template/templates/__init__.py` will be modified as

```
from flask import Flask

app = Flask(__name__,
 static_folder = './public',
 template_folder="./static")

from templates.hello.views import hello_blueprint

# register the blueprints
app.register_blueprint(hello_blueprint)
```

Next, the `hello_template/templates/hello/views.py` will be modified as

```
from flask import render_template, Blueprint

hello_blueprint = Blueprint('hello',__name__)

@hello_blueprint.route('/')
@hello_blueprint.route('/hello')
def index():
 return render_template("index.html")
```

> *We have defined a blueprint in the view file, we don't need the app object anymore here, and our complete routing is defined on a blueprint named* `hello_blueprint`*. So instead of* `@app.route`*, we used* `@hello_blueprint.route`*. The same blueprint is imported in* `hello_template/templates/__init__.py` *and registered on the application object. A rule of thumb is that each view should have its own blueprint.*

Installing and Configuring Webpack

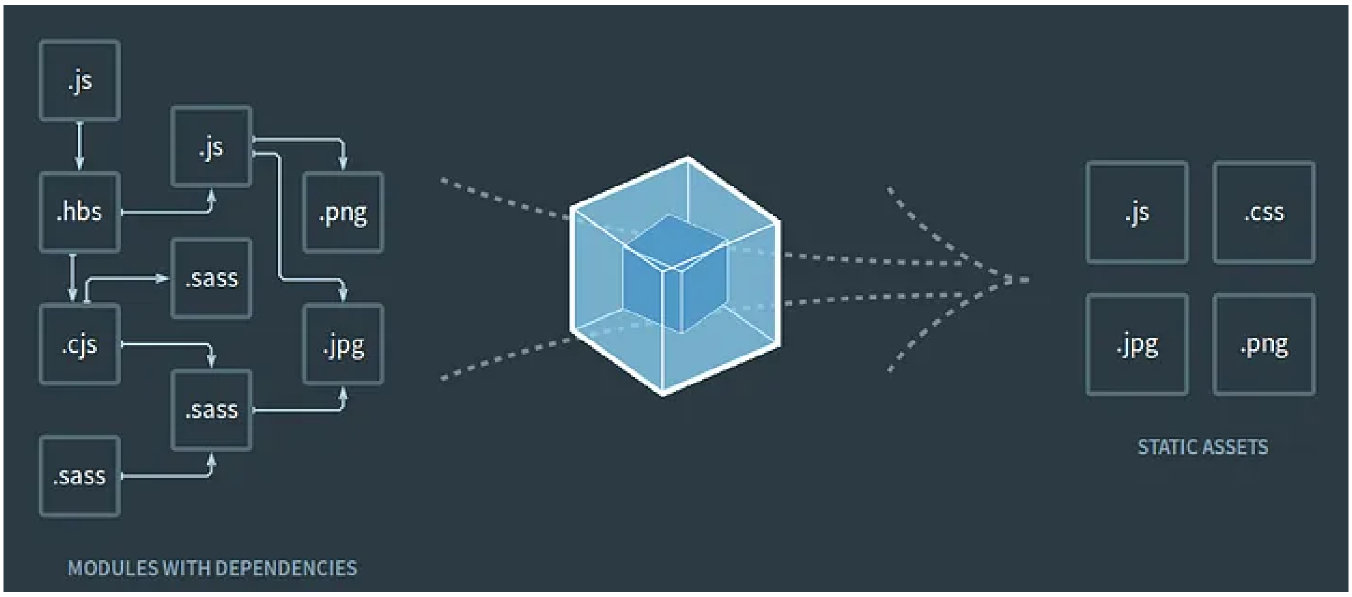


Illustration of how webpack works from its official website

webpack is a module bundler which will be used for JSX transformation and module/dependency bundling. The usage of modules helps organize JavaScript code by splitting it into multiple files, each one declaring its own dependencies. The module bundler then automatically packs everything together in the correct load order.

A lot of tools that handle this intermediary step, including Grunt, Gulp, and Brunch, among others. But in general, the React community has adopted webpack as the preferred tool for this job.

Installing Webpack: Webpack can be installed through `npm`. Install it globally using `npm install -g webpack` or add it as dependency in your project with `npm install –save-dev webpack.`

In this project, we will add webpack as dependency. So we first create a `hello_template/templates/static/package.json` file. To create this file, you can open a terminal in the `hello_template/templates/static/` folder and run the command `npm init`

> *The `package.json` file is a standard npm manifest that holds various information about the project and let the developer specify dependencies(that can get automatically downloaded and installed) and define script tasks.*

The `init` command will ask you a series of questions regarding your project(such as project name, description, information about the author, etc.).

With a package.json file in place, install **webpack** and the **webpack cli** with

`npm install --save-dev webpack && npm install -D webpack-cli`

> ***Note:*** *Every installations we will do with* `npm` *should be at a terminal opened in the* `hello_template/templates/static/` *directory where* `package.json` *resides.*

Defining a Config File: The basic command line syntax for webpack is "`webpack {entry file} {destination for bundled file}`". Webpack requires you to point only one entry file — it will figure out all the project's dependencies automatically. Additionally, if you don't have webpack installed globally like in our case, you will need to reference the webpack command in the `node_modules` folder of the project. The command will look like this:
`node_modules/.bin/webpack js/index.jsx public/bundle.js`

However, webpack has a lot of different and advance options and allows for the usage of loaders and plugins to apply transformation on the loaded modules. Although its possible to use webpack with all options from the command line as above, the process tends to get slow and error-prone. A better approach is to define a configuration file — a simple JavaScript module where you can put all information relating to your build.

Create a file `hello_template/templates/static/webpack.config.js`. It must reference the entry file and the destination for the bundled file as shown below:

```
const webpack = require('webpack');
const resolve = require('path').resolve;

const config = {
 entry: __dirname + '/js/index.jsx',
 output:{
      path: resolve('../public'),
      filename: 'bundle.js',
      publicPath: resolve('../public')

},
 resolve: {
  extensions: ['.js','.jsx','.css']
 },
};

module.exports = config;
```

> `__dirname` *is a node.js global variable containing the name of the directory that the currently executing script resides in.*
>
> *The* `bundle.js` *file created by webpack will be stored in the public folder as indicated in* `path: resolve('../public')` *above.*

Add Run Command: To make the development process more fluid, we will add some few run commands *(build, dev-build,* and *watch*) to the package.json file.

Build is used for production builds, and *dev-build* for non-minified builds. *Watch* is similar to *dev-build*, with the added benefit that it monitors your project files. Any changed files will be automatically rebuilt, and refreshing your browser will show the changes you just made.

Here is the package.json

```
{
  "name": "hello_template",
  "version": "1.0.0",
  "description": "A template for creating a full stack wep app with
Flask, NPM, Webpack, and Reactjs",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack -p --progress --config webpack.config.js",
    "dev-build": "webpack --progress -d --config webpack.config.js",
    "watch": "webpack --progress -d --config webpack.config.js --
watch"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/Eyongkevin/my-portfolio.git"
  },
  "keywords": [
    "portfolio",
    "template",
    "python",
    "react",
    "npm",
    "webpack"
  ],
  "author": "Eyong Kevin",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/Eyongkevin/my-portfolio/issues"
  },
  "homepage": "https://github.com/Eyongkevin/my-portfolio#readme",
  "devDependencies": {
    "webpack": "^4.19.1",
    "webpack-cli": "^3.1.0"
  },
}
```

> *Your `package.json` may be different from mine due to the answers you may have provided during the creating of this file with `npm init`*
>
> *You may notice some variables with my personal information like `url` , `author` and `homepage` . You may want to change it to your own personal information.*

Generating source maps: One of the most important and used configurations for webpack is Source Maps. While packing together all of your project's JavaScript modules into one(or a few) bundled file to use on the browser present a lot of advantages, one clear disadvantage is that you won't be able to reference back your original code in their original files when debugging in the browser. It becomes very challenging to locate exactly where the code you are trying to debug maps to your original authored code. This is where Source Maps comes in — A source map provides a way of mapping code within a bundled file back to its original source file, making the code readable and easier to debug in the browser.

To configure Webpack to generate source maps that points to the original files, we use the `devtool` setting. It has many options but for this project, we will be using the '`eval-source-map`' as it generates a complete source map. Though it comes with some disadvantages, we will use it only on development mode.

```
const webpack = require('webpack');
const resolve = require('path').resolve;

const config = {
 devtool: 'eval-source-map',
 entry: __dirname + '/js/index.jsx',
 output:{
      path: resolve('../public'),
      filename: 'bundle.js',
      publicPath: resolve('../public')

},
 resolve: {
  extensions: ['.js','.jsx','.css']
 },
};
```

```
module.exports = config;
```

> *There are even faster options for devtool. Although faster, these options don't map the bundled code straight to the original source files, and are more appropriate for bigger projects were build times are a concern. You can learn more about all the available options at webpack's documentation*

Loaders: One of the most exciting features of Webpack are loaders. Through the use of loaders, webpack can preprocess the source files through external scripts and tools as it loads them to apply all kinds of changes and transformations. The transformations are useful in many circumstances, for example for parsing JSON files into plain JavaScript or turning next generation's JavaScript code into regular JavaScript that current browsers can understand. Loaders can be configured under the "`modules`" key in `webpack.config.js`.

Loader configuration setting includes:

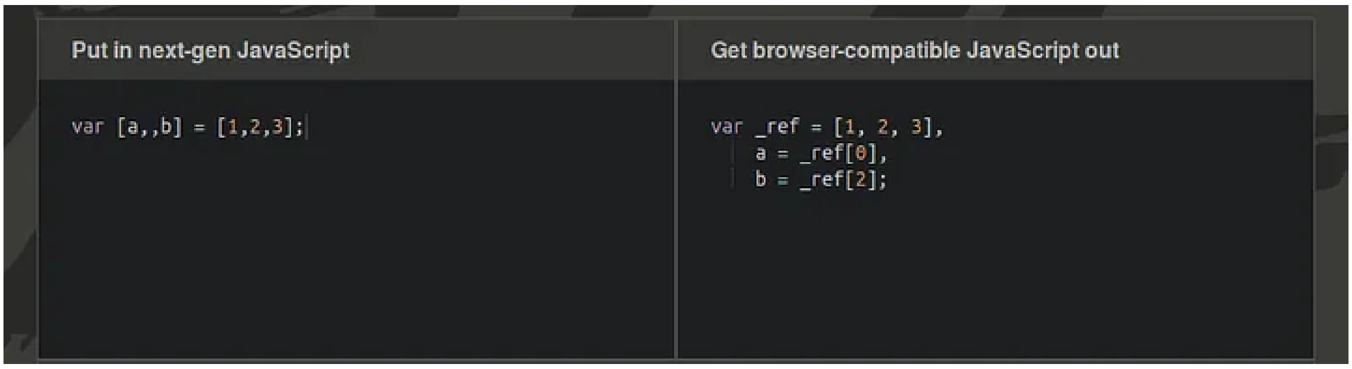
- **test**: A regular expression that matches the file extensions that should run through this loader (Required).
- **loader**: The name of the loader (Required).
- **include/exclude**: Optional setting to manually set which folder and files the loader should explicitly add or ignore.
- **query**: The query setting can be used to pass additional options to the loader.

In this project, we will use babel as our loader to transform React's JSX into plain JavaScript as below:

```
const webpack = require('webpack');

const config = {
        devtool: 'eval-source-map',
 entry: __dirname + '/js/index.jsx',
 output:{
  path: __dirname + '/dist',
  filename: 'bundle.js',

},
 resolve: {
  extensions: ['.js','.jsx','.css']
 },
 module: {
  rules: [
  {
   test: /\.jsx?/,
   loader: 'babel-loader',
   exclude: /node_modules/,
  }]
 }
};

module.exports = config;
```

> *The loader that is used in the above code is '`Babel`'. Continue below to see how to install and configure babel.*
>
> *We had excluded `node_modules` here because we won't want babel to transform any node modules hence making the loading faster.*

Add Babel Support


Put in next-gen JavaScript
Get browser-compatible JavaScript out

```
var [a,,b] = [1,2,3];
```

```
var _ref = [1, 2, 3],
    a = _ref[0],
    b = _ref[2];
```

Code example of how Babel transforms next-generation Javascript to browser-compatible Javascript, as shown on the Babel website

Babel is a platform for JavaScript compilation and tooling. It let you;

- Use next versions of JavaScript (ES6, ES2015, ES2016, etc), not yet fully supported in all browsers
- Use JavaScript syntax extensions, such as React's JSX

Though babel is a stand alone tool, we will use it in this project as a loader since it pairs very well with Webpack.

Babel is distributed in different npm modules. The core functionality is available in the "`babel-core`" npm package, the integration with webpack is available through the "`babel-loader`" npm package, and for every type of feature and extensions we want to make available to our code, we will need to install a separate package (the most common are `babel-preset-es2015` and `babel-preset-react`, for compiling ES6 and React's JSX respectively).

Open a terminal in the directory *`hello_template/templates/static/`* *and i*nstall babel and all the feature needed for this project as development dependencies with the command:
`npm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react`

Now that we have installed babel, we can enable our project to use ES6 modules and syntax, as well as JSX by adding as presets in the `webpack.config.js` file.

```
const webpack = require('webpack');

const config = {
        devtool: 'eval-source-map',
 entry: __dirname + '/js/index.jsx',
 output:{
  path: __dirname + '/dist',
  filename: 'bundle.js',

},
 resolve: {
  extensions: ['.js','.jsx','.css']
 },
 module: {
  rules: [
  {
   test: /\.jsx?/,
   loader: 'babel-loader',
   exclude: /node_modules/,
   query:{
     presets: ['react','es2015']
   }
  }]
 }
};

module.exports = config;
```

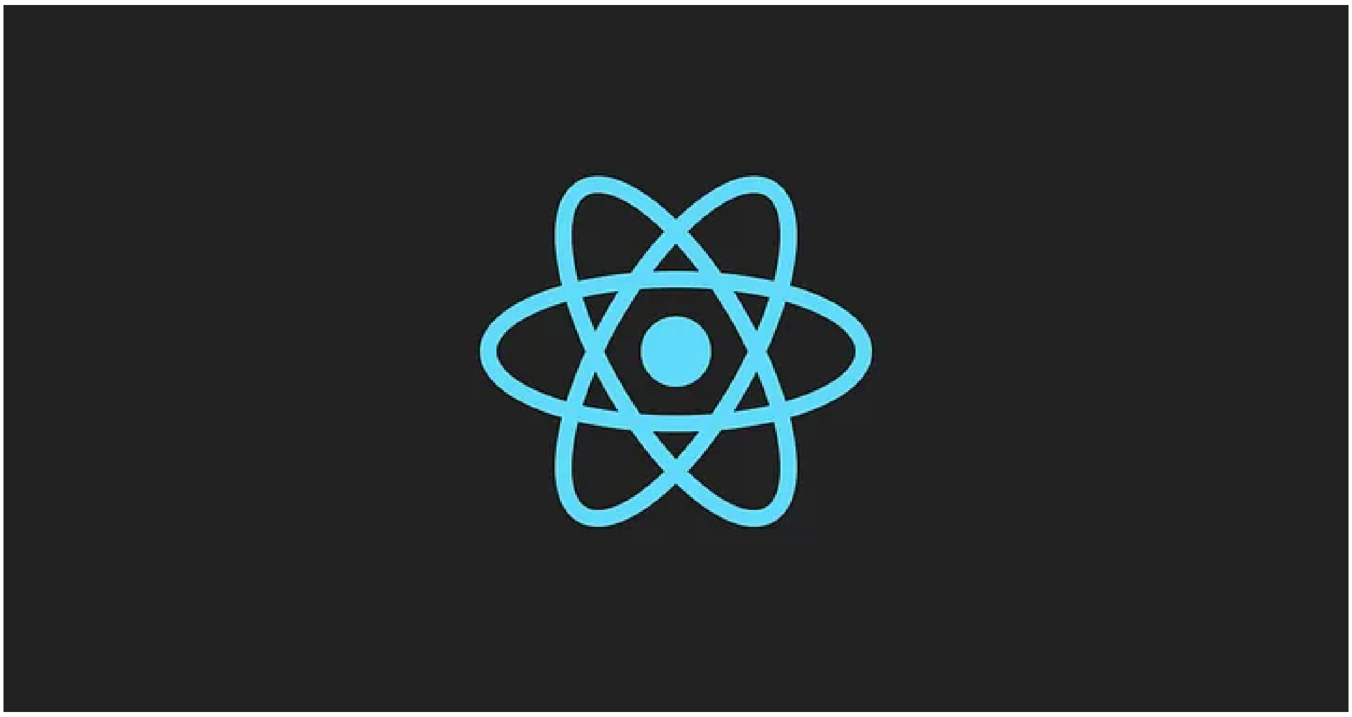
However, many developers opt to create a separate babel resource configuration- namely, a `.babelrc` file(with a leading dot). The only babel-specific configuration we have in place so far is the presents definition-which may not justify the creation of a babel-specific configuration file. But since additional webpack and babel features will be added with time, we are better off using babel configuration file.

Create a `.babelrc` file in the `hello_template/templates/static/` directory:
In Linux, you can use `gedit .babelrc`
This will create an unsaved file which you can fill in the codes below and save

```
{
 “presets”: [“react”, “es2015”]

}
```

> *Babel will look for a `.babelrc` in the current directory of the file being transpiled. If one does not exist, it will travel up the directory tree until it finds either a `.babelrc,` or a `package.json` with a `“babel”: {}` hash within. You can read more from the official website*

Add Reactjs



Reactjs logo

React is an open-source project created by Facebook. It offers a novel approach towards building user interfaces in JavaScript.

There are a lot of JavaScript MVC frameworks out there. So why did Facebook build React and why would we use it in this project?

- **Reactive Rendering is Simple**: Most JavaScript frameworks use data binding to keep the interface in sync with the state, which comes with disadvantages in maintainability, scalability, and performance. React lets you write in a declarative way how components should look and behave. And when the data changes, React conceptually renders the whole interface again. Conceptually in the sense that it uses an in-memory, lightweight representation of the DOM called "`virtual DOM`".
- **Component-Oriented Development Using Pure JavaScript:** In React application, everything is made of components, which is self-contained, concern-specific building blocks. It allows a "divide and conquer" approach where no particular part needs to be especially complex. So it's easy to create complex and more feature-rich components made of smaller components. These components are written in plain JavaScript, instead of template languages, this gives you a full-featured programming language to render views because template languages can be limiting

Install Node.js: To follow along, you will need to have Node.js installed. JavaScript was born to run on the browser, but Node.js makes it possible to run JavaScript programs on your local computer and on a server. If you don't have Node.js installed, you can install it now by downloading the installer for Windows, Mac or Linux here.

Install React: Open a terminal in `hello_template/templates/static/` and run the command

```
$ npm i react react-dom --save-dev
```

Creating .js and .jsx files: We recall that our `hello_template/templates/__init__.py` file imported our views from `hello_template/templates/hello/` folder. And the `view.py` file in the hello folder Will listen to a url and render a template file '`index.html`' which is found in the `hello_template/templates/static/` directory . This HTML file is almost empty, it simply loads the bundled javaScript and provides a Div in which to render react components.

```
<!— index.html —>
<html>
  <head>
    <meta charset="utf-8">
    <!-- Latest compiled and minified bootstrap CSS -->
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/latest/css/bootstrap
.min.css">
    <title>Creating a Full-Stack Python Application with Flask, NPM,
React.js and Webpack</title>
  </head>
  <body>
    <div id="content" />
    <script src="public/bundle.js" type="text/javascript"></script>
  </body>
</html>
```

This HTML file's location was specified in `hello_template/templates/__init__.py` with the variable `template_folder='./static`

> *The `bundle.js` file here is the webpack's single JavaScript file containing all the modules packed in the correct order. And it's location is been specified in `hello_template/templates/__init__.py` as `static_folder='./public'`*

Now we will fill the file `hello_template/templates/static/js/index.jsx` with a simple React app, and have it load the appropriate component from a router file `hello_template/templates/static/js/routes.js` into our `index.html` file

```
import React from 'react';
import ReactDOM from 'react-dom';
import routes from "./routes";

ReactDOM.render(routes, document.getElementById("content"));
```

Our `hello_template/templates/static/js/routes.js` file will do the work of calling the appropriate components and consequently displaying our "**Hello React!**" message.

Installing react router: React Router is the most popular solution for adding routing to a React application. It keeps the UI in sync with the URL by having components associated with routes (at any nesting level). When the user changes the URL, components get unmounted and mounted automatically.

Router is an external library, it must be installed with npm (along with the History library, which is a React Router peer dependency

Using npm:

```
$ npm install --save react-router-dom history
```

Add the following code into the `routes.js` file:

```
import React from 'react';
import { HashRouter, Route, hashHistory } from 'react-router-dom';
import Home from './components/Home';
// import more components
export default (
    <HashRouter history={hashHistory}>
     <div>
      <Route path='/' component={Home} />
     </div>
    </HashRouter>
);
```

> `routes.js` *imports the* `Home` *class from* `hello_template/templates/static/js/components/Home.jsx`*, so this class needs to be exported for it to be importable.*
>
> *Remember we said the* `index.html` *file contains a Div in which to render react component. We see that* `index.html` *has a Div with Id '*`content`*' in which* `index.jsx` *renders the routes from* `routes.js`

React Components: In a React application, everything is made of components, which are self-contained, concern-specific building blocks. Developing applications using components allows a "divide and conquer" approach where no particular part needs to be especially complex. They are kept small and because they can be combined, it's easy to create complex and more feature-rich components made of smaller components.

Using React components architecture comes with some advantages like:

- It enables the use of the same principles to render HTML for the Web as well as native iOS and Android view

- Events behave in a consistent, standards-compliant way in all browsers and devices, automatically using delegation.
- React components can be rendered on the server for SEO and perceived performance

So React components are just building blocks of React UIs which simply consist of a JavaScript class with a render method that returns a description of the component's UI.

In this project, we will create just one component which will describe our web app home page. However, we should create for each UI, a component which will take care of describing the specified UI.

Enter the code below into the `hello_template/templates/static/js/components/Home.jsx` file

```
import React, { Component } from 'react';
export default class Home extends Component {
    render() {
       return (
          <h1>Hello React!</h1>
       )
    }
}
```

> *We can notice the HTML tags in the middle of the JavaScript code. As mentioned, React has a syntax extension to JavaScript called JSX that lets us write XML(and consequently HTML) inline with code. Hence we had used babel to transform the JSX into plain JavaScript.*

Stylesheets: Webpack provides two loaders to deal with stylesheets: `css-loader` and `style-loader`. Each loader deals with different tasks: While the css-loader looks for `@import` and `url` statements and resolves them, the style-loader adds all the computed style rules into the page. Combined together, these loaders enable you to embed stylesheets into a Webpack JavaScript bundle.

Install css-loader and style-loader with npm

`npm install — save-dev style-loader css-loader`

In sequence, update the webpack configuration file, as shown below:

```
const webpack = require('webpack');
const config = {
        devtool: 'eval-source-map',
 entry: __dirname + '/js/index.jsx',
 output:{
  path: __dirname + '/dist',
  filename: 'bundle.js',
},
 resolve: {
  extensions: ['.js','.jsx','.css']
 },
 module: {
  rules: [
  {
   test: /\.jsx?/,
   loader: 'babel-loader',
   exclude: /node_modules/,
  },
  {
         test: /\.css$/,
         loader: 'style-loader!css-loader?modules'
  }]
}
};
module.exports = config;
```

> *The exclamation point ("!") can be used in a loader configuration to chain different loaders to the same file types*
>
> *From webpack v4, the automatic -loader module name extension has been removed. It is no longer possible to omit the `-loader` extension when referencing loaders. So instead of the former `loader: style!css` now it is `loader: 'style-loader! css-loader`*

Now lets write a simple css file just to illustrate. Copy the code below in to the file `hello_template/templates/public/css/main.css`

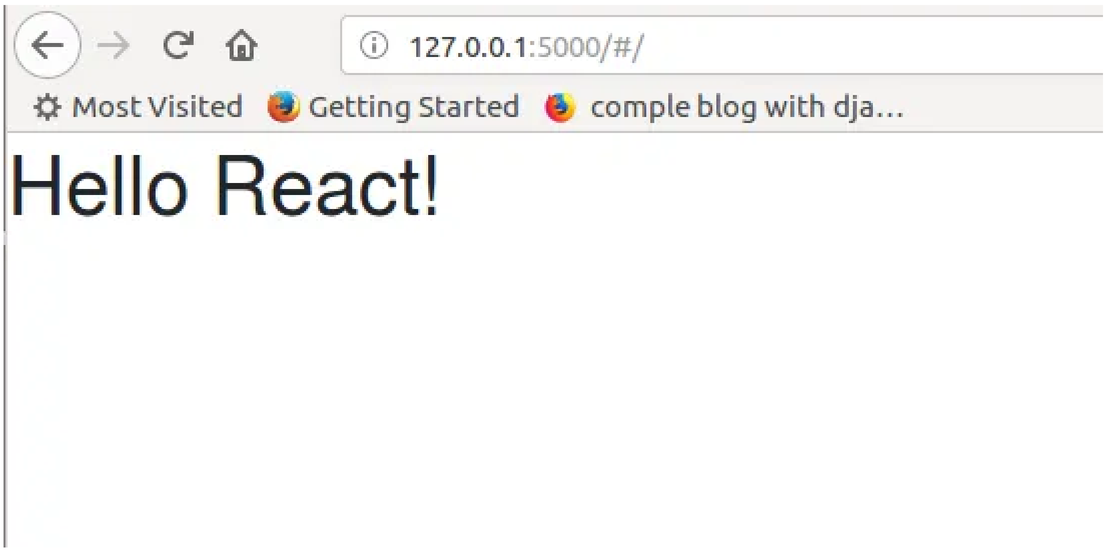
```
html {
 box-sizing: border-box;
 -ms-text-size-adjust: 100%;
 -webkit-text-size-adjust: 100%;
}
*, *:before, *:after {
 box-sizing: inherit;
}
body {
 margin: 0;
 font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
}
h1, h2, h3, h4, h5, h6, p, ul {
 margin: 0;
 padding: 0;
}
```

Finally, we should remember that Webpack starts on an entry file defined in its configuration file and build all the dependency tree by following statements like import, require, url among others. This means that the main CSS file must also be imported somewhere in the application in order for webpack to find it.

Let’s import our `main.css` into `index.html`

```
<!- index.html ->
<html>
  <head>
    <meta charset="utf-8">
    <!-- Latest compiled and minified bootstrap CSS -->
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/latest/css/bootstrap
.min.css">
    <link rel="stylesheet" href="public/css/main.css">
    <title>Creating a Full-Stack Python Application with Flask, NPM,
React.js and Webpack</title>
  </head>
  <body>
    <div id="content" />
    <script src="public/bundle.js" type="text/javascript"></script>
  </body>
</html>
```

Running the App



Hello React!

Our Web App running on Firefox Web Browser

Open a terminal in the `hello_template/templates/static/` folder and start a development watch server

```
npm run watch
```

> *If you get an error here similar to* ****npm WARN babel-loader@8.0.2 requires a peer of @babel/core@^7.0.0 but none was installed.*****, then you should downgrade your babel-loader to 7.x as follows:*
>
> `npm install babel-loader@^7 --save-dev`

Open a terminal at the root directory of this project and start the python server

```
python run.py
```

If all is working correctly, you will be given an address `http://127.0.0.1:5000/` which you can open in your favorite browser and see our application running and displaying "**Hello React!**"

And that's it. You've just created your basic full-stack application.

Thank you for your time, I would be updating this article accordingly. You can contribute by commenting, suggesting and sharing this article so that we can together, educate the world.

Also, if you like this article, you may want to continue learning by following the continuation of this article here, where we will see how to make our web application **component-oriented** and **data-driven**

I will be releasing more articles like this, so make sure you follow me here on Medium so that you won't miss any of my new releases.

JavaScript Reactjs Webpack NPM Python Flask

Follow

Written by eyong kevin

163 Followers · Writer for ITNEXT

Bachelor degree in software engineering, currently working on Machine Learning, AI, Deep Learning and React. When I am not working, I produce beats.

Recommended from Medium