****

**ReactJS**

**Submitted By:**

**Shaikh Sameer**

**Introduction**

React is a front-end library developed by Facebook. It is used for handling the view layer for web and mobile apps. ReactJS allows us to create reusable UI components. It is currently one of the most popular JavaScript libraries and has a strong foundation and large community behind it.

ReactJS is JavaScript library used for building reusable UI components. According to React official documentation, following is the definition −

React is a library for building composable user interfaces. It encourages the creation of reusable UI components, which present data that changes over time. Lots of people use React as the V in MVC. React abstracts away the DOM from you, offering a simpler programming model and better performance. React can also render on the server using Node, and it can power native apps using React Native. React implements one-way reactive data flow, which reduces the boilerplate and is easier to reason about than traditional data binding.

## React Features

* **JSX** − JSX is JavaScript syntax extension. It isn't necessary to use JSX in React development, but it is recommended.
* **Components** − React is all about components. You need to think of everything as a component. This will help you maintain the code when working on larger scale projects.
* **Unidirectional data flow and Flux** − React implements one-way data flow which makes it easy to reason about your app. Flux is a pattern that helps keeping your data unidirectional.
* **License** − React is licensed under the Facebook Inc. Documentation is licensed under CC BY 4.0.

## React Advantages

* Uses virtual DOM which is a JavaScript object. This will improve apps performance, since JavaScript virtual DOM is faster than the regular DOM.
* Can be used on client and server side as well as with other frameworks.
* Component and data patterns improve readability, which helps to maintain larger apps.

## React Limitations

* Covers only the view layer of the app, hence you still need to choose other technologies to get a complete tooling set for development.
* Uses inline templating and JSX, which might seem awkward to some developers.

# react\_webpack

to run this project steps:

npm install npm start

for production npm build

# What is webpack?

webpack is one of the pillars of modern Web Development. It’s an incredibly powerful tool. webpack ingests raw React components for producing Javascript code that (almost) every browser can understand.

for more details refer following link:

<https://webpack.js.org/>

steps to integrate webpack with raw structure:

1. mkdir react\_webpack && cd react\_webpack
2. create minimal directory

mkdir -p src

1. Initialize

npm init -y

1. Install webpack

npm i webpack --save-dev

1. Will Also need webpack cli

npm i webpack-cli --save-dev

1. Add Following to package.json

"scripts": {

"build": "webpack --mode production"

}

As it is raw structure we need to add webpack loader:

Since the browser can’t understand React components as they come there is the need for some kind of transformation.

Required following:

i. babel preset env for compiling Javascript ES6 code down to ES5 (please note that babel-preset-es2015 is now deprecated)

ii. babel preset react for compiling JSX and other stuff down to Javascript

npm i @babel/core babel-loader @babel/preset-env @babel/preset-react --save-dev

Configure babel, Create a new file named .babelrc inside the project folder

{

"presets": ["@babel/preset-env", "@babel/preset-react"]

}

1. Add React Dependencies

npm i react react-dom --save-dev

create directories

mkdir -p src/js/components/{container,presentational}

1. Create File inside container AppLayout.js

import React, { Component } from "react";

class AppLayout extends Component {

constructor() {

super();

this.state = {

title: ""

};

}

render() {

return (

<div>test</div>

);

}

}

export default AppLayout;

1. To display our React form we must tell Webpack to produce an HTML page. The resulting bundle will be placed inside a <script></script>tag.

create src\index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/css/bootstrap.min.css" >

<title>React\_Webpack</title>

</head>

<body>

<div class="container">

<div class="row mt-5">

<div class="col-md-4 offset-md-1">

<div id="root">

<!-- form -->

</div>

</div>

</div>

</div>

</body>

</html>

Webpacks needs two additional components for processing HTML: html-webpack-plugin and html-loader.

npm i html-webpack-plugin html-loader --save-dev

1. Create file webpack.config.js at base directory

add following configuration to same:

const HtmlWebPackPlugin = require("html-webpack-plugin");

module.exports = {

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader"

}

},

{

test: /\.html$/,

use: [

{

loader: "html-loader"

}

]

}

]

},

plugins: [

new HtmlWebPackPlugin({

template: "./src/index.html",

filename: "./index.html"

})

]

};

1. Webpack expects entry point index.js

add file to src\index.js

import React, { Component } from "react";

import ReactDOM from "react-dom";

import AppLayout from "./js/components/container/appLayout";

const wrapper = document.getElementById("root");

wrapper ? ReactDOM.render(<AppLayout />, wrapper) : false;

1. npm run build

Configuring Development Server for above implementation

1. npm i webpack-dev-server --save-dev
2. open packeg.json and update

"scripts": {

"start": "webpack-dev-server --open --mode development",

"build": "webpack"

}

1. npm start

## Using JSX

JSX looks like a regular HTML in most cases. We already used it in the Environment Setup chapter. Look at the code from **App.jsx** where we are returning **div**.

### App.jsx

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

Hello World!!!

</div>

);

}

}

export default App;

Even though it's similar to HTML, there are a couple of things we need to keep in mind when working with JSX.

## Nested Elements

If we want to return more elements, we need to wrap it with one container element. Notice how we are using **div** as a wrapper for **h1**, **h2** and **p**elements.

### App.jsx

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<h1>Header</h1>

<h2>Content</h2>

<p>This is the content!!!</p>

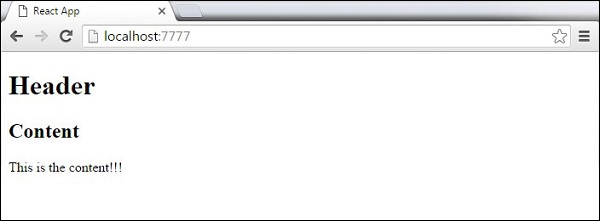
</div>

);

}

}

export default App;



## Attributes

We can use our own custom attributes in addition to regular HTML properties and attributes. When we want to add custom attribute, we need to use **data-**prefix. In the following example, we added **data-myattribute** as an attribute of **p** element.

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<h1>Header</h1>

<h2>Content</h2>

<p data-myattribute = "somevalue">This is the content!!!</p>

</div>

);

}

}

export default App;

## JavaScript Expressions

JavaScript expressions can be used inside of JSX. We just need to wrap it with curly brackets **{}**. The following example will render **2**.

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<h1>{1+1}</h1>

</div>

);

}

}

export default App;



We cannot use **if else** statements inside JSX, instead we can use **conditional (ternary)** expressions. In the following example, variable **i** equals to **1** so the browser will render **true**, If we change it to some other value, it will render **false**.

import React from 'react';

class App extends React.Component {

render() {

var i = 1;

return (

<div>

<h1>{i == 1 ? 'True!' : 'False'}</h1>

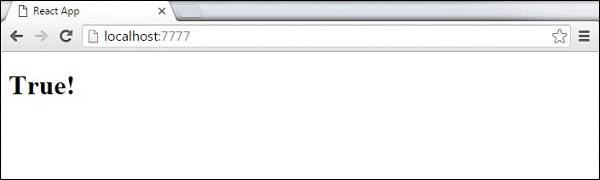
</div>

);

}

}

export default App;



## Styling

React recommends using inline styles. When we want to set inline styles, we need to use **camelCase** syntax. React will also automatically append **px** after the number value on specific elements. The following example shows how to add **myStyle** inline to **h1** element.

import React from 'react';

class App extends React.Component {

render() {

var myStyle = {

fontSize: 100,

color: '#FF0000'

}

return (

<div>

<h1 style = {myStyle}>Header</h1>

</div>

);

}

}

export default App;



## Comments

When writing comments, we need to put curly brackets **{}** when we want to write comment within children section of a tag. It is a good practice to always use **{}** when writing comments, since we want to be consistent when writing the app.

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<h1>Header</h1>

{//End of the line Comment...}

{/\*Multi line comment...\*/}

</div>

);

}

}

export default App;

## Stateless Example

Our first component in the following example is **App**. This component is owner of **Header** and **Content**. We are creating **Header** and **Content** separately and just adding it inside JSX tree in our **App** component. Only **App** component needs to be exported.

### App.jsx

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<Header/>

<Content/>

</div>

);

}

}

class Header extends React.Component {

render() {

return (

<div>

<h1>Header</h1>

</div>

);

}

}

class Content extends React.Component {

render() {

return (

<div>

<h2>Content</h2>

<p>The content text!!!</p>

</div>

);

}

}

export default App;

To be able to render this on the page, we need to import it in **main.js** file and call **reactDOM.render()**. We already did this while setting the environment.

### main.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';

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

The above code will generate the following result.



## Stateful Example

In this example, we will set the state for owner component (**App**). The **Header** component is just added like in the last example since it doesn't need any state. Instead of content tag, we are creating **table** and **tbody** elements, where we will dynamically insert **TableRow** for every object from the **data**array.

It can be seen that we are using EcmaScript 2015 arrow syntax (=>) which looks much cleaner than the old JavaScript syntax. This will help us create our elements with fewer lines of code. It is especially useful when we need to create a list with a lot of items.

### App.jsx

import React from 'react';

class App extends React.Component {

constructor() {

super();

this.state = {

data:

[

{

"id":1,

"name":"Foo",

"age":"20"

},

{

"id":2,

"name":"Bar",

"age":"30"

},

{

"id":3,

"name":"Baz",

"age":"40"

}

]

}

}

render() {

return (

<div>

<Header/>

<table>

<tbody>

{this.state.data.map((person, i) => <TableRow key = {i}

data = {person} />)}

</tbody>

</table>

</div>

);

}

}

class Header extends React.Component {

render() {

return (

<div>

<h1>Header</h1>

</div>

);

}

}

class TableRow extends React.Component {

render() {

return (

<tr>

<td>{this.props.data.id}</td>

<td>{this.props.data.name}</td>

<td>{this.props.data.age}</td>

</tr>

);

}

}

export default App;

### main.js

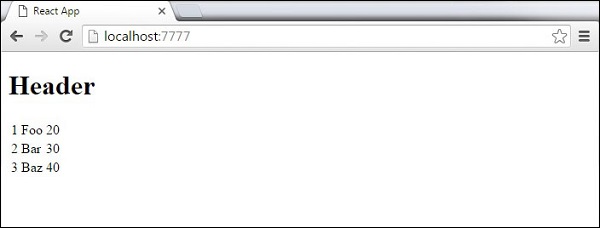
import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';

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

**Note** − Notice that we are using **key = {i} inside map()** function. This will help React to update only the necessary elements instead of re-rendering the entire list when something changes. It is a huge performance boost for larger number of dynamically created elements.



## Using Props

The following sample code shows how to create a stateful component using EcmaScript2016 syntax.

### App.jsx

import React from 'react';

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

header: "Header from state...",

content: "Content from state..."

}

}

render() {

return (

<div>

<h1>{this.state.header}</h1>

<h2>{this.state.content}</h2>

</div>

);

}

}

export default App;

### main.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';

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

This will produce the following result.



## Using Props

When we need immutable data in our component, we can just add props to **reactDOM.render()** function in **main.js** and use it inside our component.

### App.jsx

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<h1>{this.props.headerProp}</h1>

<h2>{this.props.contentProp}</h2>

</div>

);

}

}

export default App;

### main.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';

ReactDOM.render(<App headerProp = "Header from props..." contentProp = "Content

from props..."/>, document.getElementById('app'));

export default App;

This will produce the following result.



## Validating Props

In this example, we are creating **App** component with all the **props** that we need. **App.propTypes** is used for props validation. If some of the props aren't using the correct type that we assigned, we will get a console warning. After we specify validation patterns, we will set **App.defaultProps**.

### App.jsx

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<h3>Array: {this.props.propArray}</h3>

<h3>Bool: {this.props.propBool ? "True..." : "False..."}</h3>

<h3>Func: {this.props.propFunc(3)}</h3>

<h3>Number: {this.props.propNumber}</h3>

<h3>String: {this.props.propString}</h3>

<h3>Object: {this.props.propObject.objectName1}</h3>

<h3>Object: {this.props.propObject.objectName2}</h3>

<h3>Object: {this.props.propObject.objectName3}</h3>

</div>

);

}

}

App.propTypes = {

propArray: React.PropTypes.array.isRequired,

propBool: React.PropTypes.bool.isRequired,

propFunc: React.PropTypes.func,

propNumber: React.PropTypes.number,

propString: React.PropTypes.string,

propObject: React.PropTypes.object

}

App.defaultProps = {

propArray: [1,2,3,4,5],

propBool: true,

propFunc: function(e){return e},

propNumber: 1,

propString: "String value...",

propObject: {

objectName1:"objectValue1",

objectName2: "objectValue2",

objectName3: "objectValue3"

}

}

export default App;

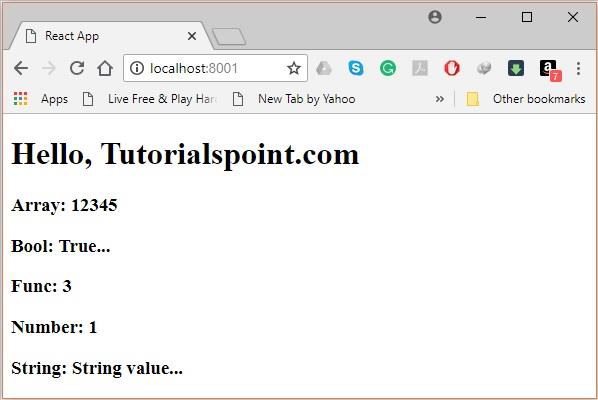
### main.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';

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



**Router**

## Step 1 - Install a React Router

A simple way to install the **react-router** is to run the following code snippet in the **command prompt** window.

C:\Users\username\Desktop\reactApp>npm install react-router

## Step 2 - Create Components

In this step, we will create four components. The **App** component will be used as a tab menu. The other three components **(Home), (About)** and **(Contact)** are rendered once the route has changed.

### main.js

import React from 'react';

import ReactDOM from 'react-dom';

import { Router, Route, Link, browserHistory, IndexRoute } from 'react-router'

class App extends React.Component {

render() {

return (

<div>

<ul>

<li>Home</li>

<li>About</li>

<li>Contact</li>

</ul>

{this.props.children}

</div>

)

}

}

export default App;

class Home extends React.Component {

render() {

return (

<div>

<h1>Home...</h1>

</div>

)

}

}

export default Home;

class About extends React.Component {

render() {

return (

<div>

<h1>About...</h1>

</div>

)

}

}

export default About;

class Contact extends React.Component {

render() {

return (

<div>

<h1>Contact...</h1>

</div>

)

}

}

export default Contact;

## Step 3 - Add a Router

Now, we will add routes to the app. Instead of rendering **App** element like in the previous example, this time the **Router** will be rendered. We will also set components for each route.

### main.js

ReactDOM.render((

<Router history = {browserHistory}>

<Route path = "/" component = {App}>

<IndexRoute component = {Home} />

<Route path = "home" component = {Home} />

<Route path = "about" component = {About} />

<Route path = "contact" component = {Contact} />

</Route>

</Router>

), document.getElementById('app'))

When the app is started, we will see three clickable links that can be used to change the route.



**References**

* <https://www.tutorialspoint.com/reactjs>
* <https://www.valentinog.com/blog/react-webpack-babel/>
* <https://www.robinwieruch.de/minimal-react-webpack-babel-setup/>
* <https://medium.com/@siddharthac6/getting-started-with-react-js-using-webpack-and-babel-66549f8fbcb8>