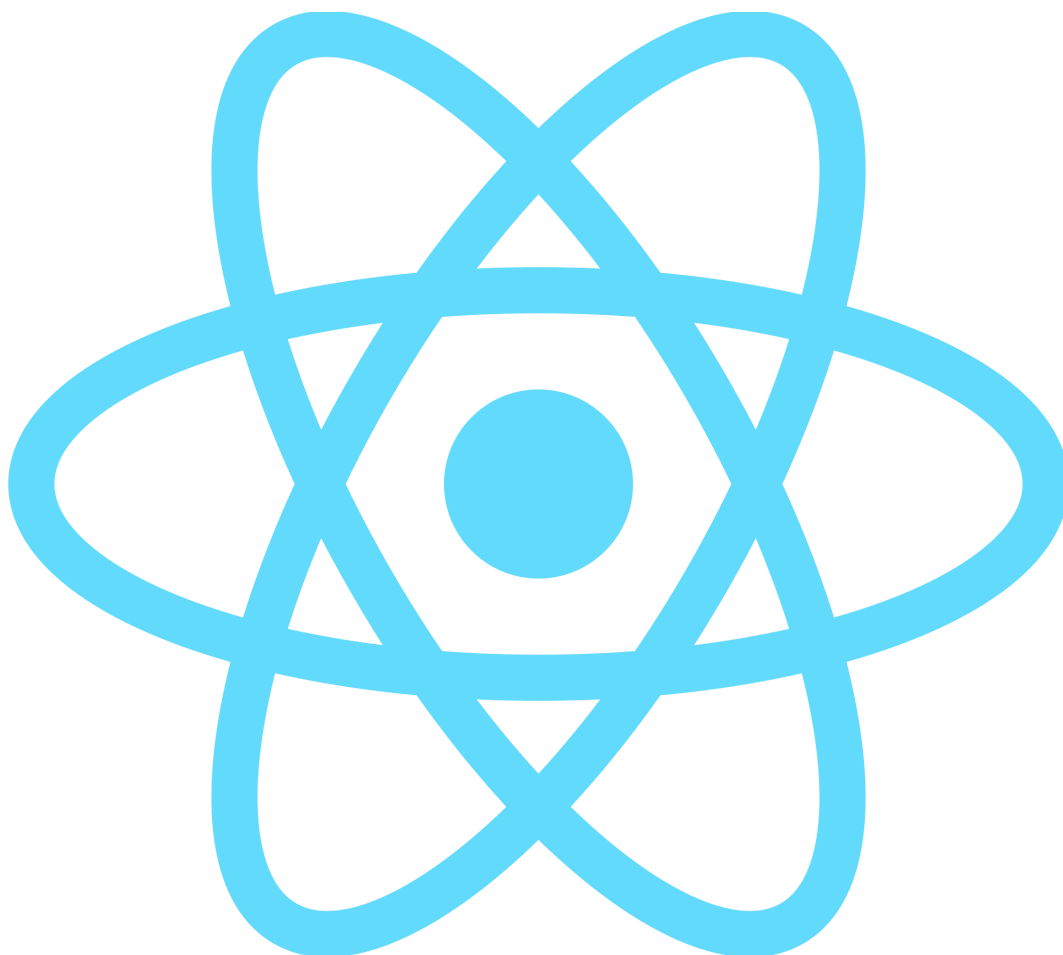


Agenda:

- What is React ?
- Why use React instead of other frameworks ?
- Difference between Single Page Application and Multiple Page Application
- What is Real DOM(Server DOM), Browser DOM and Virtual DOM ?
- How Virtual DOM works in React - React Architecture ?
- Introduction to NPM and NPX
- Node.js Installation
- Different ways to create React project
- Building the first react project
- React project structure

Let's start our first session with React and we will start from scratch, starting from:

What is React ?



React.js was released by a software engineer working for Facebook – Jordane Walke in 2011. React is a Javascript Library focused on creating declarative user interfaces (UIs) using a component-based concept. It's used for handling the view layer and can be used for web and mobile app. React's main goal is to be extensive, fast, declarative, flexible, and simple.

React is not a framework, it is specifically a library. The explanation for this is that React only deals with rendering UIs and reserves many things at the discretion of individual projects. The standard set of tools for creating an application using ReactJS is frequently called the **stack**.

React is a library for helping developers build user interfaces (UIs) as a tree of small pieces called components. A *component* is a mixture of HTML or JavaScript that captures all of the logic required to display a small section of a larger UI. Each of these components can be built up into successive complex parts of an app.

Now after getting the overview of React, let's see why react is more popular or chosen over other frameworks:

Why use React instead of other frameworks ?

So the main question is why you should choose ReactJS as a frontend development stack while there are a lot of others. Here are some reasons:

- **Speedless.** React allows developers to use individual parts of their application on both the client and server sides, and any changes they make will not affect the application's logic. This makes the development process extremely fast.
- **Components support.** The use of HTML tags and JS codes makes it easy to work with a huge dataset containing DOM. React acts as an intermediary that represents the DOM and helps you decide which component requires changes to get accurate results.
- **Easy to use and learn.** ReactJS is incredibly user-friendly and makes any UI interactive. It also allows you to quickly and efficiently build applications, which is time-saving for clients and developers alike.
- **SEO Friendly.** A common problem complained by most web developers is that traditional Javascript Frameworks often have problems with SEO. ReactJS solves this problem by helping developers navigate different search engines easily through the fact that the ReactJS application can run on the server, and the virtual DOM renders and returns it to the browser as a web page^{**}.
- **One-way Data Binding.** One-way data-binding implies that absolutely anyone can trace all the changes that have been made to a segment of the data. This is also one of the reasons that makes React so easy.

Just to the gist of the popularity of the React, following are the companies that uses React:

- [Facebook](#)
- [Atlassian](#)
- [Discord](#)
- [Uber Eats](#)
- [Netflix](#)
- [Airbnb](#)
- [Snapchat](#)
- [Trello](#)

So, now you can think of the popularity of React framework.

Now, before going deep with React, let's first learn about Single page and Multiple page applications:

Single Page application vs Multiple Page application

Web applications are unwittingly replacing the old desktop applications. They are more convenient to use, they are easy to update, and they are not bound to one device. And even though users are gently moving from browser-based web applications into the mobile ones, the demand for complex and refined apps is already huge and is still growing. If you are thinking about creating your own application, you've probably heard that there are two main design patterns for web apps: multi-page application (MPA) and single-page application (SPA). And of course, both models have their pros and cons.

Before you start turning your idea into the real application, you have to answer a bunch of important questions. To decide what app model is better for you, you should follow content-first approach, which emphasizes the importance of putting your application content before everything else. That's because content is the main reason for which users will or won't use the application. And this leads us to the most important questions: what content do you want to present and what content your users will care about the most.

Single Page Application

A single-page application is an app that works inside a browser and does not require page reloading during use. You are using this type of application every day. These are, for instance: Gmail, Google Maps, Facebook or GitHub. SPAs are all about serving an outstanding UX by trying to imitate "natural" environment in the browser — no page reloads, no extra wait time. It is just one web page that you visit which then loads all other content using JavaScript — which they heavily depend on.

SPA requests the markup and data independently and renders pages straight in the browser. We can do this thanks to the advanced JavaScript frameworks like React JS, AngularJS, Ember.js, Meteor.js, Knockout.js. Single-page sites help keep the user in one, comfortable web space where content is presented to the user in a simple, easy and workable fashion.

Pros of a Single Page Application :

- SPA is fast, as most resources (HTML+CSS+Scripts) are only loaded once throughout the lifespan of application. Only data is transmitted back and forth.
- The development is simplified and streamlined. There is no need to write code to render pages on the server. It is much easier to get started because you can usually kick off development from a file://URI, without using any server at all.
- SPAs are easy to debug with Chrome, as you can monitor network operations, investigate page elements and data associated with it.
- It's easier to make a mobile application because the developer can reuse the same backend code for web application and native mobile application.

- SPA can cache any local storage effectively. An application sends only one request, store all data, then it can use this data and works even offline.

Multiple Page Application

Multiple-page applications work in a “traditional” way. Every change eg. display the data or submit data back to server requests rendering a new page from the server in the browser. These applications are large, bigger than SPAs because they need to be. Due to the amount of content, these applications have many levels of UI. Luckily, it’s not a problem anymore. Thanks to AJAX, we don’t have to worry that big and complex applications have to transfer a lot of data between server and browser. That solution improves and it allows to refresh only particular parts of the application. On the other hand, it adds more complexity and it is more difficult to develop than a single-page application.

There are of course some cons in SPA and some pros in MPA, but as we can determine from above explanation that SPA is anyone’s first choice, and that is where React comes really handy as it solves or removes out most of complexity in building SPA’s which we would see as we go along with React.

Now, let’s learn one of the concepts which makes the React that much popular i.e., the concept of Virtual DOM:

In order to understand the virtual DOM, let’s first see the What is Real DOM ?

Real DOM

First things first, DOM stands for “Document Object Model”. The DOM in simple words represents the UI of your application. Everytime there is a change in the state of your application UI, the DOM gets updated to represent that change. Now the catch is frequently manipulating the DOM affects performance, making it slow.

What makes DOM manipulation slow?

The DOM is represented as a tree data structure. Because of that, the changes and updates to the DOM are fast. But after the change, the updated element and its children have to be re-rendered to update the application UI. The re-rendering or re-painting of the UI is what makes it slow. Therefore the more UI components you have, the more expensive the DOM updates could be, since they would need to be re-rendered for every DOM update.

Virtual DOM

That’s where the concept of virtual DOM comes in and performs significantly better than the real DOM. The virtual DOM is only a virtual representation of the DOM. Everytime the state of our application changes, the virtual DOM gets updated instead of the real DOM.

Well, you may ask ” Isn’t the virtual DOM doing the same thing as the real DOM, this sounds like double work? How can this be faster than just updating the real DOM?”

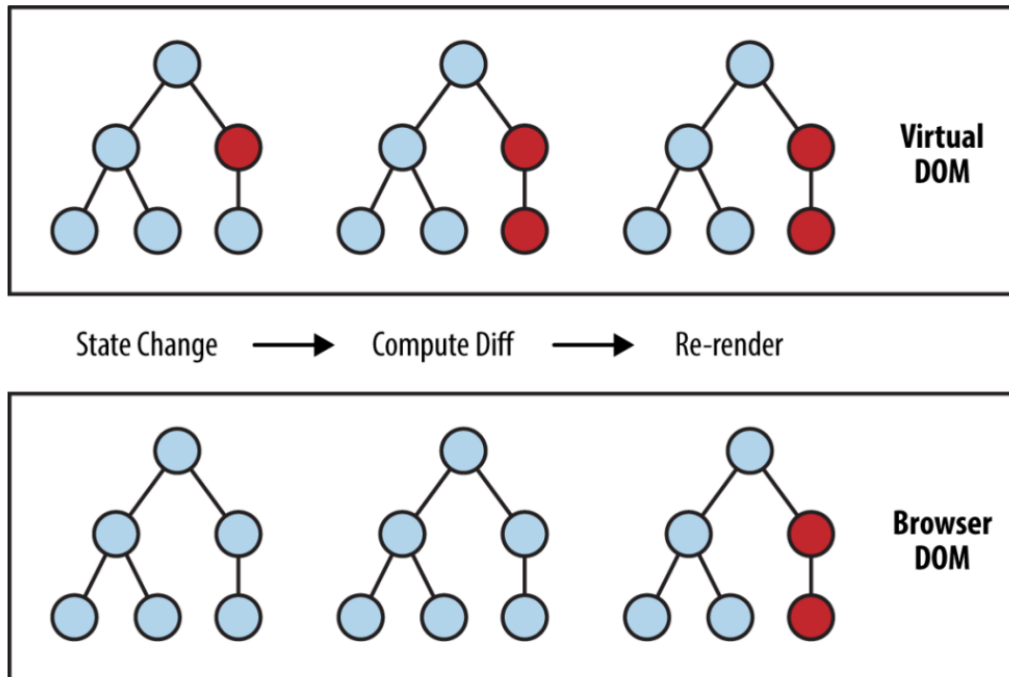
The answer is virtual DOM is much faster and efficient, here is why.

How is Virtual DOM faster?

When new elements are added to the UI, a virtual DOM, which is represented as a tree is created. Each element is a node on this tree. If the state of any of these elements changes, a new virtual DOM tree is created. This tree is then compared or “diffed” with the previous virtual DOM tree.

Once this is done, the virtual DOM calculates the best possible method to make these changes to the real DOM. This ensures that there are minimum operations on the real DOM. Hence, reducing the performance cost of updating the real DOM.

The image below shows the virtual DOM tree and the diffing process.



The red circles represent the nodes that have changed. These nodes represent the UI elements that have had their state changed. The difference between the previous version of the virtual DOM tree and the current virtual DOM tree is then calculated. The whole parent subtree then gets re-rendered to give the updated UI. This updated tree is then batch updated to the real DOM.

How does React use Virtual DOM

Now that you have a fair understanding of what a Virtual DOM is, and how it can help with performance of your app, let's look into how React leverages the virtual DOM.

In React every UI piece is a component, and each component has a state. React follows the observable pattern and listens for state changes. When the state of a component changes, React updates the virtual DOM tree. Once the virtual DOM has been updated, React then compares the current version of the virtual DOM with the previous version of the virtual DOM. This process is called "diffing".

Once React knows which virtual DOM objects have changed, then React updates **only** those objects, in the real DOM. This makes the performance faster when compared to manipulating the real DOM directly. This makes React stand out as a high performance JavaScript library.

In simple words, you tell React what state you want the UI to be in, and it makes sure that the DOM matches that state. The great benefit here is that as a developer, you would not need to know how the attribute manipulation, event handling or the manual DOM updates happen behind the scenes.

All of these details are abstracted away from React developers. All you need to do is update the states of your component as and when needed and React takes care of the rest. This ensures a superior developer experience when using React.

Architecture:

In a **Model View Controller(MVC) architecture**, React is the 'View' responsible for how the app looks and feels.

MVC is an architectural pattern that splits the application layer into Model, View, and Controller. The model relates to all data-related logic; the view is used for the UI logic of the application, and the controller is an interface between the Model and View.

Now let's see how we can create our first react project, but before this we need to learn about Node package manager:

Introduction to NPM and NPX:

NPM :

npm(node package manager) is regarded as the standard package manager used in javascript. The npm registry crossed a million packages. It is the largest single-language code repository. That being said, it is a little obvious now that **npm is a big deal**.

npm is automatically installed when you install Node.js

Since it's a package manager, it is used to manage downloads and handle dependencies of your project.

It is a common saying that

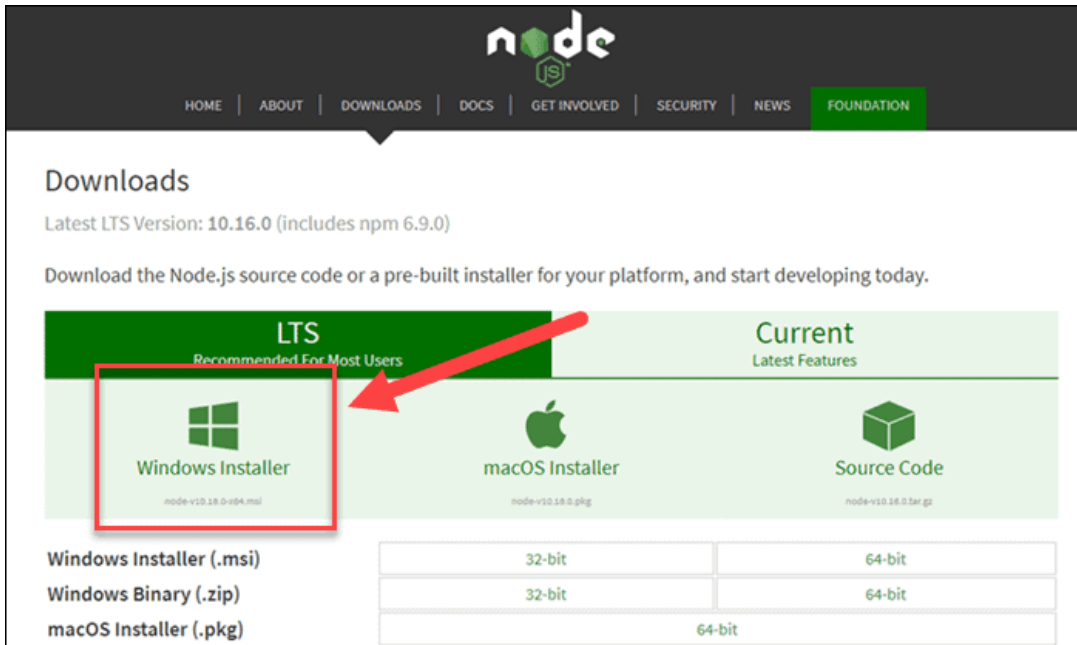
There is a package for everything in npm

So, as we have learned just that npm is installed automatically with Node.js, So, let's first see that how we can install Node.js :

Installation of Node.js on Windows:

Step 1: Download Node.js installer

In a web browser, navigate to <https://nodejs.org/en/download/>. Click the **Windows Installer** button to download the latest default version. At the time the article was written, version 10.16.0-x64 was the latest version. The Node.js installer includes the NPM package manager.



Step 2: Install Node.js and NPM from browser

1. Once the installer finishes downloading, launch it. Open the **downloads** link in your browser and click the file. Or, browse to the location where you have saved the file and double-click it to launch.
2. The system will ask if you want to run the software – click **Run**.
3. You will be welcomed to the Node.js Setup Wizard – click **Next**.
4. On the next screen, review the license agreement. Click **Next** if you agree to the terms and install the software.
5. The installer will prompt you for the installation location. Leave the default location, unless you have a specific need to install it somewhere else – then click **Next**.
6. The wizard will let you select components to include or remove from the installation. Again, unless you have a specific need, accept the defaults by clicking **Next**.
7. Finally, click the **Install** button to run the installer. When it finishes, click **Finish**.

Step 3: Verify Installation

Open a command prompt (or PowerShell), and enter the following:

```
node -v
```



The system should display the Node.js version installed on your system. You can do the same for NPM:

```
npm -v
```



Installing Node.js on macbook:

Step 1: Download the Node.js .pkg installer

As our first step, we need to actually *get* the official installer for Node.js on macOS. To do so, we can head over to the [Node.js Downloads page](https://nodejs.org/en/download/)

download the installer.

You can get the macOS installer by clicking the **Macintosh Installer** option - this will download the **.pkg** installer for Node.js. Make sure you save it somewhere that you'll be able to access it!

Step 2: Run the Node.js Installer

Now that you've got the installer downloaded, you'll need to run it. The installer is a pretty typical interface - it won't take long to get through it (under minute), even though there are a few parts to it. You can get through it by following the guide below:

- Introduction
- Select **Continue**
- License
- Select **Continue**
- Select **Agree**
- Installation Type
- Select **Install**
- Authenticate with your Mac to allow the Installation
- Select **Install Software**
- Summary
- Select **Close**

Step 3: Verify that Node.js was Properly installed

To verify that Node.js was installed correctly on your Mac, you can run the following command in your terminal:

```
$ node --version
```



If Node.js was properly installed, you'll see something close to (but probably not *exactly*) this:

```
$ npm --version
```



As now we are able to install the Node.js and hence npm so, let's continue our discussion on npm:

npm is written entirely in JavaScript(JS) and was developed by **Isaac Schlueter**. It was initially used to download and manage dependencies, but it has since also been used frequently in frontend JavaScript.

npm can manage packages that are local dependencies of a particular project, as well as globally-installed JavaScript tools. In addition to package downloads, npm also manages versioning, so you can install any version, higher or lower according to the needs of your project. If no version is mentioned, the latest version of the package is installed.

How to use

→ If **package.json** file exists in your project directory, all you need to do is use this command

```
npm install
```



This command will initialize the **node_modules** folder and install all packages that the project needs.

And if you need to update the installed packages, hit

```
npm update
```



All packages will be updated to their latest versions.

→ If you just need to install a single package, you can use this command

```
npm install <package_name>
```



Similarly, if you just need to update a single package, all you need to do is

```
npm update <package_name>
```



Note: By default, packages are installed in the local scope. If you need to install the package at global scope, you need to mention the flag **-g**

```
npm install <package_name> -g
```



This will install the package at the global scope in the system directory.

NPX:

NPX is an NPM package runner that makes it really easy to install any sort of node executable that would have normally been installed using NPM.

Why use NPX?

There are a number of ways to install node packages, you can have them *sitting locally* (local to the project) or *install globally* (in the user environment).

Sometimes, instead of using either of the two install methods above, you may just want to use the package and go.

Sometimes, you might just want to experiment with a list of packages as you may not know exactly what you need.

In these cases, instead of installing locally or globally, you can go straight to running those packages with NPX.

How does it work?

NPX comes bundled with NPM starting with version 5.2+. So, if your version of NPM is 5.2 or higher, then you have NPX installed.

When you run a package using NPX, it searches for the package in the local and global registry, and then it runs the package.

If the package is not already installed, NPX downloads the package files and installs the package, but it will only cache the files instead of saving it.

To use NPX, you would run a command like this:

```
npx some-package
```



One great way for you to see how quickly NPX works is to create a react app using:

```
$ npx create-react-app my-app
```



The above command will generate a react app, named **my-app**, in the path that the command was run in using the **create-react-app** package. NPX then searches for the package in your environment. If it is not found, NPX downloads the files and runs the command to create a new react app using just that one line of command.

Different ways to create a React app :

There are different ways through which we can create a React app which are as follows:

The

The simplest way to create a React application is by adding it to an existing HTML page via a **<script>** tag. This method requires only a few lines of code and no build tools.

You can follow along with a pre-existing website or create an empty HTML file to practice with.

Add an Empty Div to Your HTML Page

The first thing you need to do is open the HTML page you wish to edit and add an empty **<div>** element.

It looks like this:



```
<!-- Existing HTML of your page -->
```

```
<div id="react-component-container"></div>
```

```
<!-- Existing HTML of your page -->
```

Notice that we give the empty `<div>` a unique `id` attribute. This will help our JavaScript find the right place to put the React code in the next steps.

Add the Script Tags

For React to work, we need to pull it into the webpage. We can do that directly in our HTML page by using

Add these three `<script>` tags right before the `</body>` tag on your page:

```
<!-- Existing HTML of your page -->

<script src="https://unpkg.com/react@16/umd/react.development.js"
        crossorigin></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"
        crossorigin></script>

<script src="react-component.js"></script>
</body>
```

The top two load React from a [CDN](#). And the third one will load our React component after we create it in the next section.

If you ever move this code into production, make sure you replace `"development.js"` with `"production.min.js"` in the two top script tags.

Create the React Component

In the same folder as the HTML page you've been editing, create a file called `react-component.js` :

```
$ touch react-component.js
```

Then, add this code to it:

```
"use strict"

const e = React.createElement

class YourComponent extends React.Component {
  render() {
    return e(
      "h1",
      null,
      "Your React Component",
    )
  }
}

const domContainer = document.querySelector("#react-component-container")
ReactDOM.render(e(YourComponent), domContainer)
```

The `YourComponent` class creates a HTML element using React.

The bottom two lines find the `<div>` we created with an `id` of `"react-component-container"` and displays it in our HTML page.

You now have a React application in just a few steps and a couple of lines of code.

By setting up our own Boilerplate :

Firstly, create a project folder in which you want to create the react app.

Then initialize your project folder with npm and git

```
npm init
git init
```



Let's quickly create a .gitignore file to ignore the following folders

```
node_modules
build
```



Now, let's look at what are the basic dependencies that are needed to run a React app.

react and react-dom

These are the only two runtime dependencies you need.

```
npm install react react-dom --save
```



Transpiler(Babel)

Transpiler converts ECMAScript 2015+ code into a backward-compatible version of JavaScript in current and older browsers. We also use this transpile JSX by adding presets.

```
npm install @babel/core @babel/preset-env @babel/preset-react --save-dev
```



A simple babel config for a React app looks like this. You can add this config in .babelrc file or as a property in package.json.

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



You can add various presets and plugins based on your need.

Bundler(Webpack)

Bundler bundles your code and all its dependencies together in one bundle file(or more if you use code splitting).

```
npm install webpack webpack-cli webpack-dev-server babel-loader css-loader style-loader html-webpack-plugin --save-dev
```



A simple webpack.config.js for React application looks like this.

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: 'bundle.js',
  },
  resolve: {
    modules: [path.join(__dirname, 'src'), 'node_modules'],
  }
}
```



```

    alias: {
      react: path.join(__dirname, 'node_modules', 'react'),
    },
  },
  module: {
    rules: [
      {
        test: /\.?(js|jsx)$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
        },
      },
      {
        test: /\.css$/,
        use: [
          {
            loader: 'style-loader',
          },
          {
            loader: 'css-loader',
          },
        ],
      },
    ],
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
    }),
  ],
];
};

```

You can add various loaders based on your need.

That is all the dependencies we need. Now let's add an HTML template file and a react component.

Let's create src folder and add index.html

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>React Boilerplate</title>
</head>
<body>
  <div id="root"></div>
</body>
</html>

```



Let's create a HelloWorld.js react component in the src folder

```

import React from 'react';

const HelloWorld = () => {
  return (
    <h3>Hello World</h3>
  );
};

export default HelloWorld;

```



Let's add index.js file to the src folder

```
import React from 'react';
import { render } from 'react-dom';

import HelloWorld from './HelloWorld';

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



Finally, let's add the start and build scripts in package.json

```
"scripts": {
  "start": "webpack-dev-server --mode=development --open --hot",
  "build": "webpack --mode=production"
}
```



That is it. Now our react app is ready to run. Try the commands `npm start` and `npm run build`.

Now let's see the most popular and easy way to create React app and the one which we will be using everytime:

Creating our first React app using `create-react-app` :

We will use `npm`, a package runner tool that comes with npm 5.2+ and higher, to run

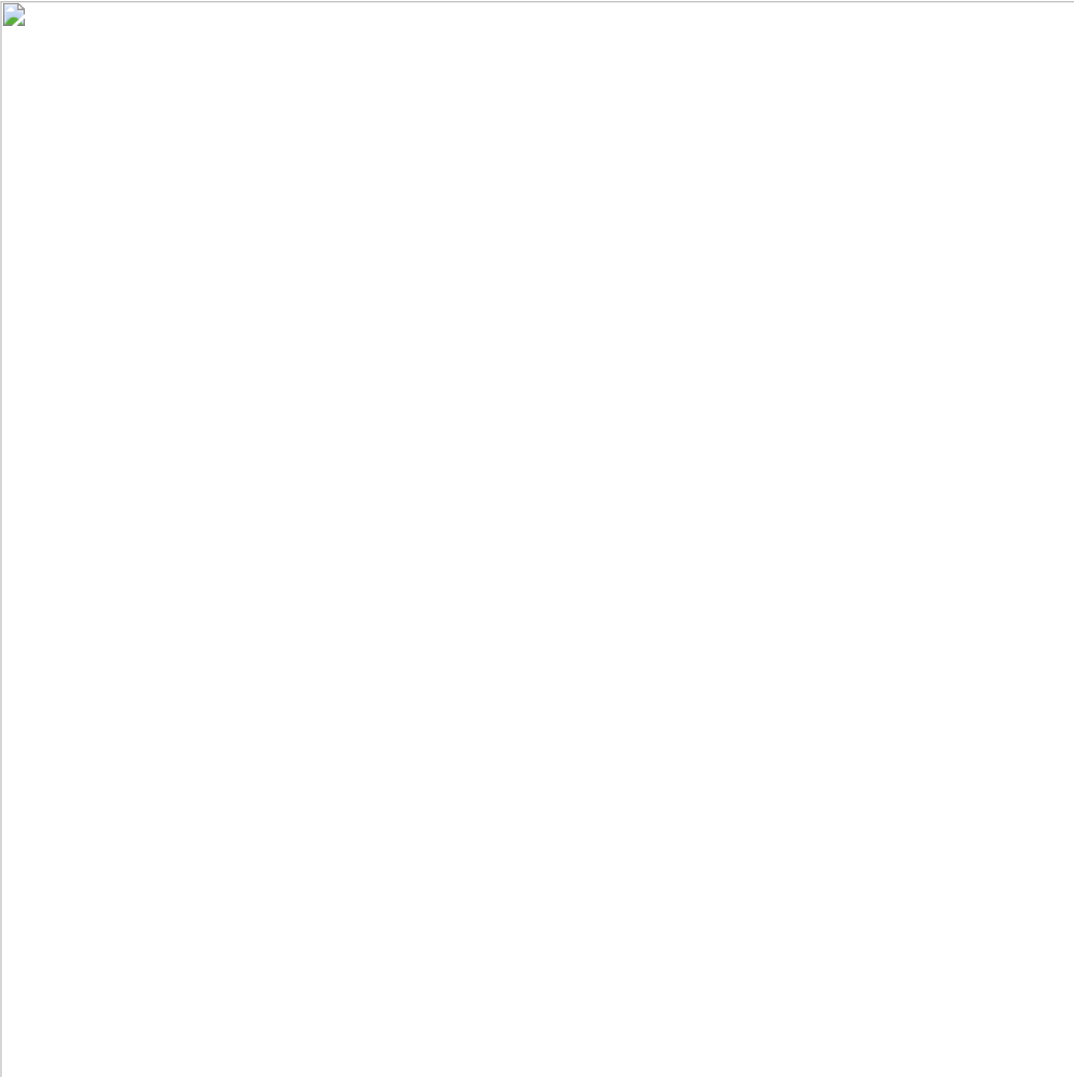
`create-react-app`.

So, open your terminal :

Just simply run this command in your terminal : `npx create-react-app myfirstreactapp`

Note: `myfirstreactapp` is just the name of the folder in which we want to initialise our react project.(Feel free to replace `myfirstreactapp` with whatever name you want, as long as it doesn't contain capital letters :-)).

Upon completion, you will get some quick tips on how to use the application:



Now, after you get the above layout on your screen then open your folder in which you have initialised the react project which in our case is **myfirstreactapp** in Visual Studio Code.

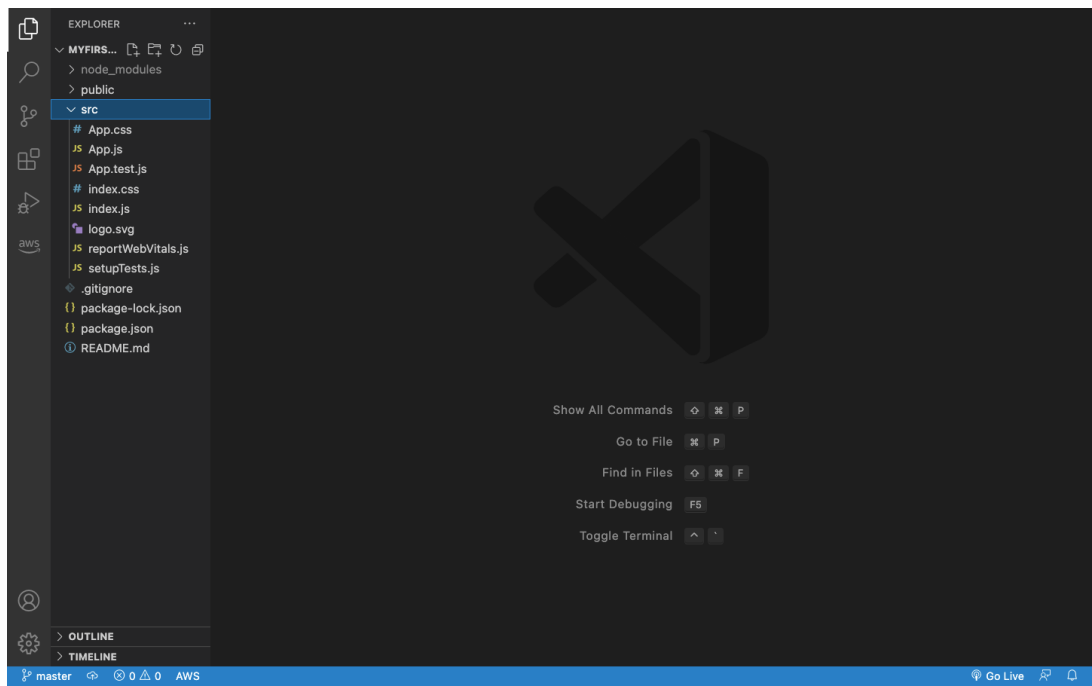
There are different ways to open the folder in your visual studio code:

- You can either drag and drop the folder into the visual studio code window.
- You can run following commands in your terminal:

```
cd myfirstreactapp
code .
```

This will also open the folder in your visual studio code.

Now, when the folder is opened in visual studio code, you will see the folder structure as shown below:



So, let's now learn the project structure of our first react app:

React Project Structure

```
myfirstreactapp
├── node_modules
├── public
│   ├── favicon.ico
│   ├── index.html
│   ├── logo192.png
│   ├── logo512.png
│   ├── manifest.json
│   └── robots.txt
├── src
│   ├── App.css
│   ├── App.js
│   ├── App.test.js
│   ├── index.css
│   ├── index.js
│   ├── logo.svg
│   ├── serviceWorker.js
│   └── setupTests.js
├── .gitignore
├── package.json
├── package-lock.json
└── README.md
```

create-react-app has taken care of setting up the main structure of the application as well as a couple of developer settings. Most of what you see will not be visible to the visitor of your web app. React uses a tool called *webpack* which transforms the directories and files here into static assets. Visitors to your site are served those static assets.

Don't worry if you don't understand too much about webpack for now. One of the benefits of using create-react-app to set up our React application is that we're able to bypass any sort of manual configuration for webpack.

Now, let's see every folder and file one by one in detail :

.gitignore

This is the standard file used by the source control tool git to determine which files and directories to ignore when committing code. While this file exists, create-react-app did not create a git repo within this folder. If you take a look at the file, it has taken care of ignoring a number of items (even **.DS_Store** for Mac users):

```
✓ ~/Documents/Projects/myfirstreactapp
[23:20 $ cat .gitignore
# See https://help.github.com/ignore-files/ for more about ignoring files.

# dependencies
/node_modules

# testing
/coverage

# production
/build

# misc
.DS_Store
.env.local
.env.development.local
.env.test.local
.env.production.local

npm-debug.log*
yarn-debug.log*
yarn-error.log*
```

package.json

```

1  {
2    "name": "myfirstreactapp",
3    "version": "0.1.0",
4    "private": true,
5    "dependencies": {
6      "@testing-library/jest-dom": "^4.2.4",
7      "@testing-library/react": "^9.5.0",
8      "@testing-library/user-event": "^7.2.1",
9      "react": "^16.13.1",
10     "react-dom": "^16.13.1",
11     "react-scripts": "3.4.3"
12   },
13   "scripts": {
14     "start": "react-scripts start",
15     "build": "react-scripts build",
16     "test": "react-scripts test",
17     "eject": "react-scripts eject"
18   },
19   "eslintConfig": {
20     "extends": "react-app"
21   },
22   "browserslist": {
23     "production": [
24       ">0.2%",
25       "not dead",
26       "not op_mini all"
27     ],
28     "development": [
29       "last 1 chrome version",
30       "last 1 firefox version",
31       "last 1 safari version"
32     ]
33   }
34 }
35

```

This file outlines all the settings for the React app.

- **name** is the name of your app
- **version** is the current version
- **"private": true** is a failsafe setting to avoid accidentally publishing your app as a public package within the npm ecosystem.
- **dependencies** contains all the required Node modules and versions required for the application. In the picture above, you'll see six dependencies. The first three, as you may have guessed, are for the purpose of testing. The next two dependencies allow us to use **react** and **react-dom** in our JavaScript. Finally, **react-scripts** provides a useful set of development scripts for working with React. In the screenshot above, the **react** version specified is **^16.13.1**. This means that npm will install the most recent major version matching 16.x.x. In contrast, you may also see something like **~1.2.3** in **package.json**, which will only install the most recent minor version matching 1.2.x.
- **scripts** specifies aliases that you can use to access some of the react-scripts commands in a more efficient manner. For example, running **npm test** in your command line will run **react-scripts test --env=jsdom** behind the scenes.
- You will also see two more attributes, **eslintConfig** and **browserslist**. Both of these are Node modules having their own set of values. **browserslist** provides information about browser compatibility of the app, while **eslintConfig** takes care of the code linting.

node_modules

This directory contains dependencies and sub-dependencies of packages used by the current React app, as specified by **package.json**. If you take look, you may be surprised by how many there are.

Running **ls -1 | wc -l** within the **node_modules/** directory will yield more than 800 subfolders. This folder is automatically added the **.gitignore** for good reason! Don't worry, even with all these dependencies, the basic app will only be around 50 KB after being minified and compressed for production.

package-lock.json

This file contains the exact dependency tree installed in **node_modules/**. This provides a way for teams working on private apps to ensure that they have the same version of dependencies and sub-dependencies. It also contains a history of changes to **package.json**, so you can quickly look back at dependency changes.

public

This directory contains assets that will be served directly without additional processing by webpack. **index.html** provides the entry point for the web app. You will also see a favicon (header icon) and a **manifest.json**.

The manifest file configures how your web app will behave if it is added to an Android user's home screen (Android users can "shortcut" web apps and load them directly from the Android UI).

src

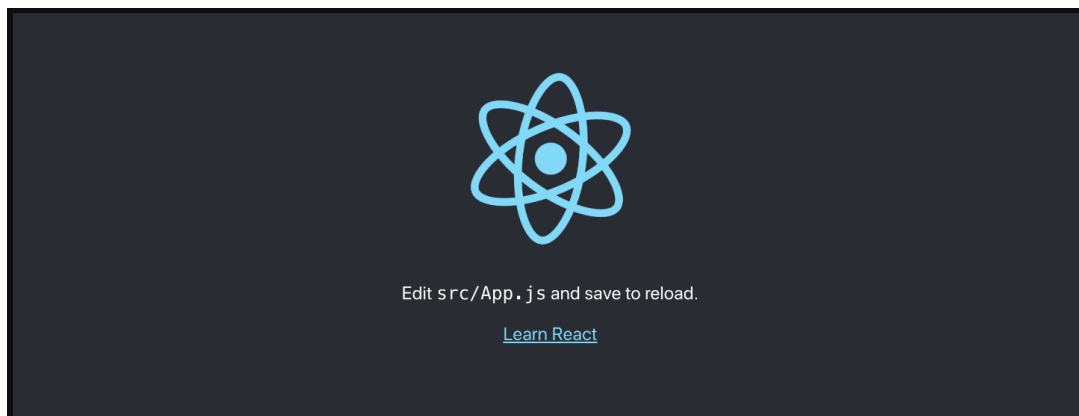
This contains the JavaScript that will be processed by webpack and is the heart of the React app. Browsing this folder, you see the main App JavaScript component (**App.js**), its associated styles (**App.css**), and test suite (**App.test.js**). **index.js** and its styles (**index.css**) provide an entry into the App and also kick off the **registerServiceWorker.js**. This service worker takes care of caching and updating files for the end-user. It allows for offline capabilities and faster page loads after the initial visit.

As your React app grows, it is common to add a **components/** directory to organize components and component-related files and a **views/** directory to organize React views and view-related files.

Now, let's start our React app development server:

Starting the React App Development Server :

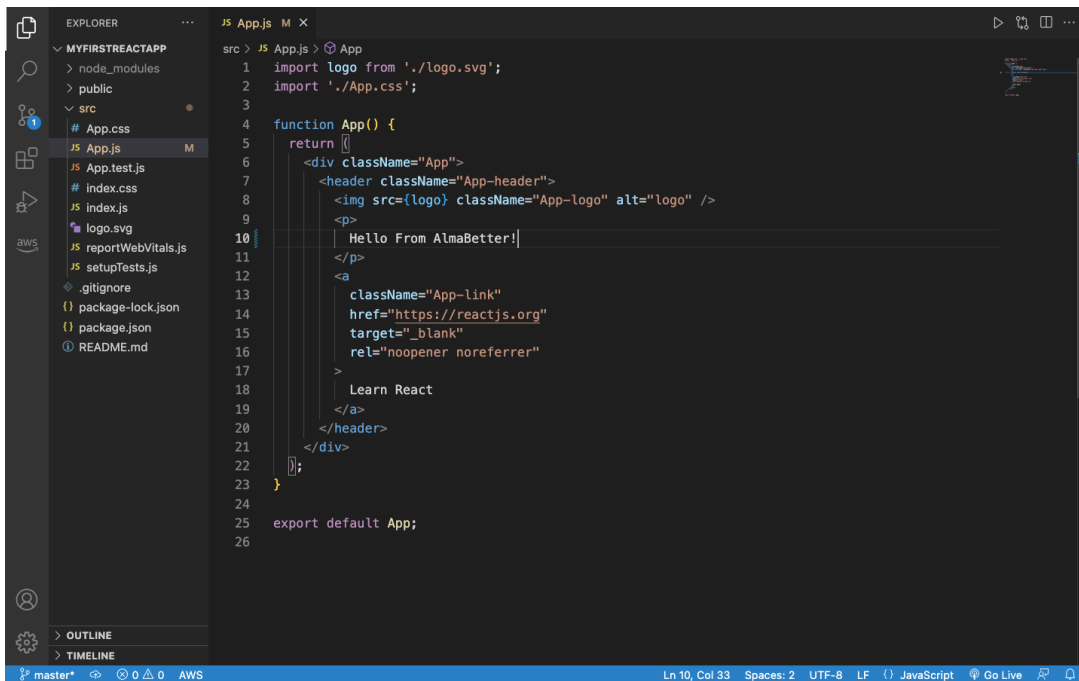
As was stated in the success message when you ran **create-react-app**, you just need to run **npm start** in your app directory to begin serving the development server. It should auto-open a tab in your browser that points to **http://localhost:3000/** (if not, manually visit that address). You will find yourself looking at a page resembling the following image:



As stated, any changes to the source code will live-update here. Let's see that in action.

Leave the current terminal tab running (it's busy serving the React app) and open **src/App.js** in your favorite text editor. You'll see what looks like a mashup of JavaScript and HTML. This is **JSX**, which is how React adds XML syntax to JavaScript. It provides an intuitive way to build React components and is compiled to JavaScript at runtime. We'll delve deeper into this in other content, but for now, let's make a simple edit and see the update in the browser.

So, open App.js and change the main paragraph text to read **Hello From AlmaBetter!** and save the file as shown below :

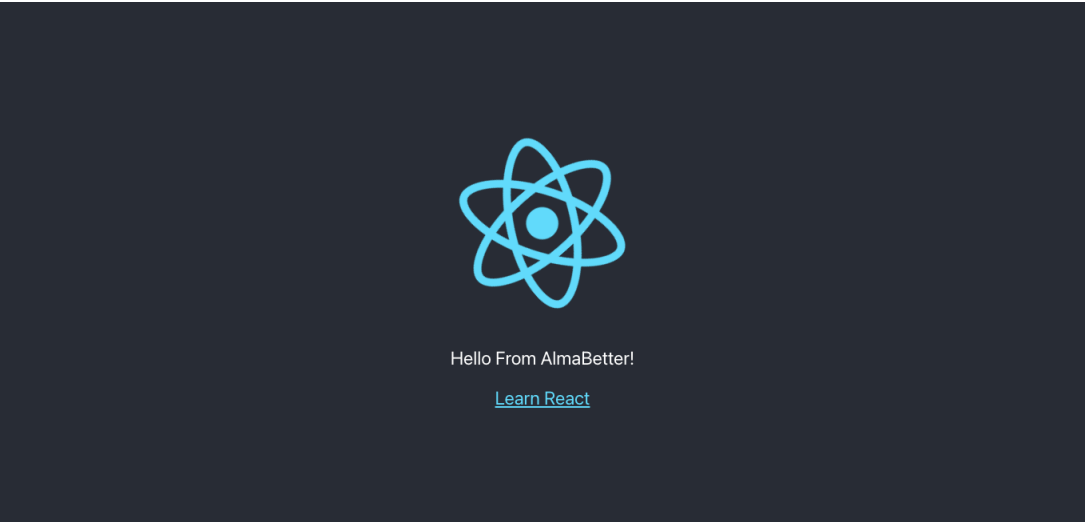


```
src > JS App.js > App
1 import logo from './logo.svg';
2 import './App.css';
3
4 function App() {
5   return (
6     <div className="App">
7       <header className="App-header">
8         <img src={logo} className="App-logo" alt="logo" />
9         <p>
10           Hello From AlmaBetter!
11         </p>
12         <a
13           className="App-link"
14           href="https://reactjs.org"
15           target="_blank"
16           rel="noopener noreferrer"
17         >
18           Learn React
19         </a>
20       </header>
21     </div>
22   );
23 }
24
25 export default App;
26
```

To save file after changing you can press:

- `cmd + s` if you are using a macbook
- `ctrl + s` if you are using a windows

Now, If you left the terminal running, you should be able to switch over to your browser and see the update as shown below :



Congratulations! You're now up and running with React. You can now begin adding functionality for your application.

Thank You !