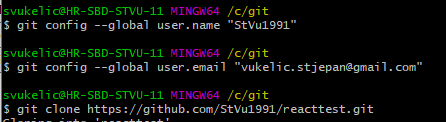
# REACT – INTRODUCTION

## REACT.JS INSTALLATION

Here is how we install REACT globally on our machine:

## CLONING GITHUB REPOSITORY



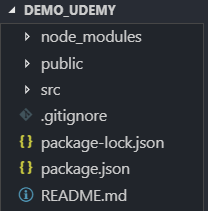
## START REACT PROJECT

Here is how we start new REACT project called 'demo1':



## STRUCTURE OF REACT PROJECT

### Main default folders

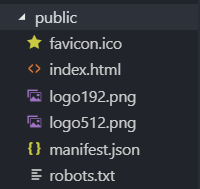


#### node\_modules

Contain all packages, all library dependencies which React need to work

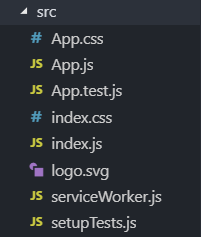
#### Public

Most important file in this folder is index.html, which is responsible for displaying your application inside browser



#### src

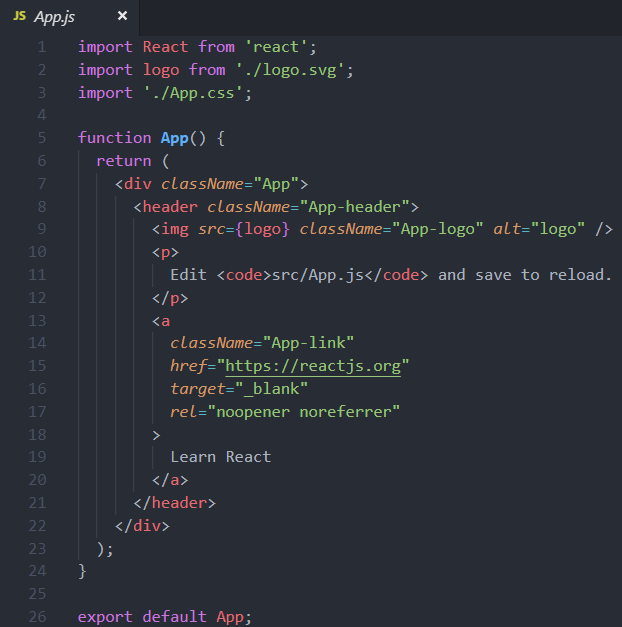
contains whole content of React application



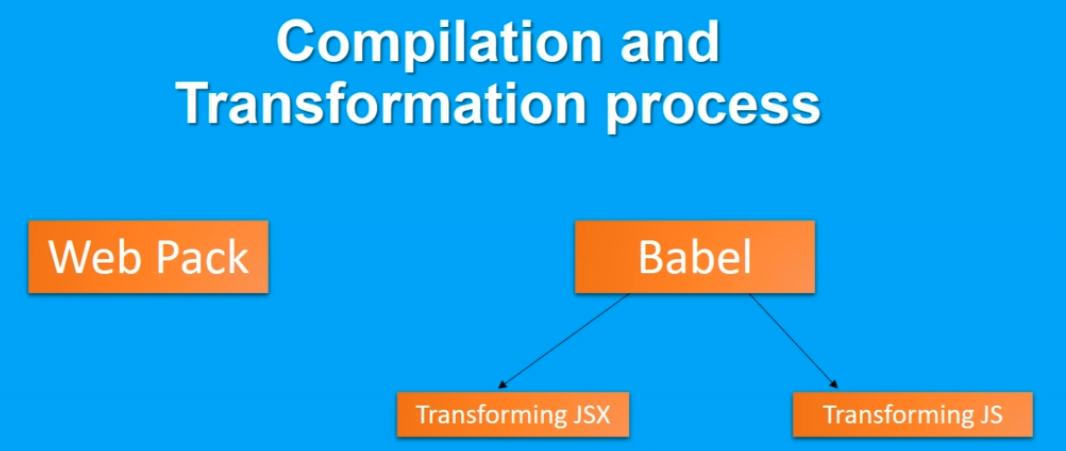
### Main default files

#### App.js

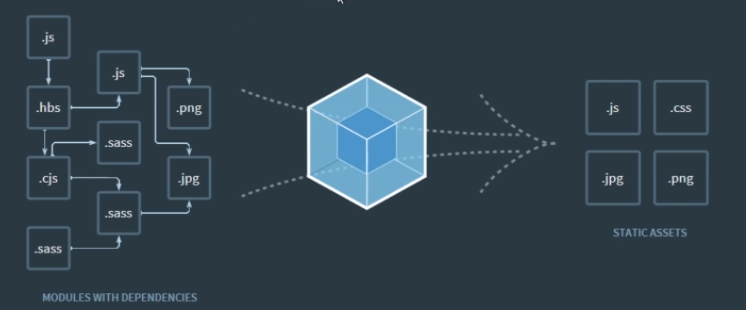
App.js placed in src folder is a example of default React component. App.js is a main building block or in other words – React component.



App.js basically contain a React component called App. Components are main building blocks of React application. App.js is a JS file, but it allows HTML code to be embedded inside Javascript. App.js also define a render() method and it is called by React to get the content to be displayed to user. To understand how it works, it is important to understand whole compilation and transformation process.



Web pack – module handler. Open-source JavaScript module bundler. In our React application it is responsible for the build up process.

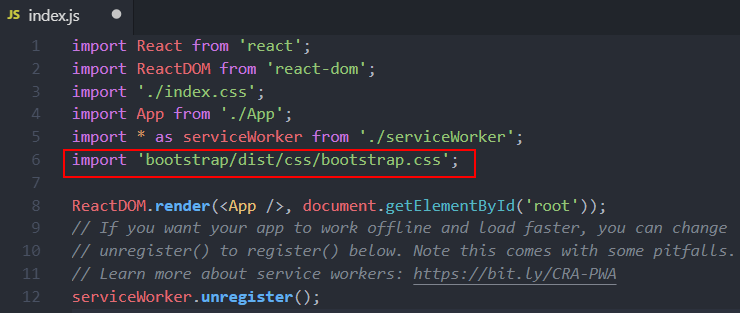


Babel – purpose of Babel is – 1. Transforming JSX 2. Transforming JS

* JSX – superset of JavaScript which allow HTML to be mixed with regular code statements. JSX is converted into calls to React API by Babel.

#### index.js

Index.js is responsible for configuring and starting a React application. When you, for example, want to add Bootstrap to your application you will add it inside this file.



## RUN REACT APPLICATION IN BROWSER

Go in folder where your application is located and run following command:



By default you should open your application entering following URL in your browser – localhost:3000.

# DYNAMIC CONTENT IN REACT APP

## Example of binding state data into render method:

export default class App extends Component {

constructor(*props*){

*super*(props);

this.state = {

userName : "Stjepan"

}

}

**render**()

{

return(

<div>

<h4 *className*="bg-primary text-white text-center p-2">

{this.state.userName} React Course

</h4>

</div>

);

}

}

### Constructor

Constructor

* constructor is a special method, it is called once component is initialized.
* **If you don’t initialize state and you don’t bind methods, you don’t need to implement a constructor for your React component.**
* The constructor for a React component is called before it is mounted. When implementing the constructor for a React.Component subclass, you should call super(props) before any other statement. Otherwise, this.props will be undefined in the constructor, which can lead to bugs.

Typically, in React constructors are only used for two purposes:

* Initializing [local state](https://reactjs.org/docs/state-and-lifecycle.html) by assigning an object to this.state.
* Binding [event handler](https://reactjs.org/docs/handling-events.html) methods to an instance.
* You **should not call setState()** in the constructor()
* Constructor is the only place where you should assign this.state directly. In all other methods, you need to use this.setState() instead
* **Avoid copying props into state! This is a common mistake:**
* constructor(props) {
* super(props);
* // Don't do this!
* this.state = { color: props.color };
* }

you can use this.props.color directly instead

## Change state data

* React app is based on changes of data
* React responds to these changes of data by calling render method again – this causes expression to be re-evaluated and using new state data afterwards

constructor(*props*){

*super*(props);

this.state = {

userName : "Stjepan",

course : "React from Zero to Hero"

}

}

**changeStateData** = () => {

this.**setState**(

{userName: this.state.userName == "Stjepan" ? "Dinosaur" : "Stjepan"}

)

}

**render**()

{

return(

<div>

<h4 *className*="bg-primary text-white text-center p-2">

<button *className*="btn btn-danger m-2” *onClick*={this.changeStateData}>

Change the State {this.state.userName}

</button>

</h4>

</div>

);

}

## Arrow functions in React app

In above example you can see that we don’t have return keyword inside changeStateData method. It is achieved using arrow function syntax ‚=>‘.

It is also possible to use arrow function sytax to use render method without return keyword.

Render with return:

**render**()

{

return(

<div>

<h4 *className*="bg-primary text-white text-center p-2">

<button *className*="btn btn-danger m-2" *onClick*={this.changeStateData}>

Change the State {this.state.userName}

</button>

</h4>

</div>

);

}

Render with arrow syntax instead of return:

**render** = () =>

<div>

<h4 *className*="bg-primary text-white text-center p-2">

<button *className*="btn btn-danger m-2" *onClick*={this.changeStateData}>

Change the State {this.state.userName}

</button>

</h4>

</div>

## Arrays of objects, array manipulations with React & rendering

export default class App extends Component {

constructor(*props*){

*super*(props);

this.state = {

userName : "Stjepan",

todoItems : [

{action:"Buy a flowers", done:false},

{action:"Do workout", done:true},

{action:"Study React", done:false},

{action:"Call a friend", done:true}],

newToDoItemText:" "

}

}

**updateNewToDoItemText** = (*event*) => {

this.**setState**({newToDoItemText : event.target.value});

}

**createNewToDoTask** = () =>

{

if (!this.state.todoItems.find(*item* => item.action === this.state.newToDoItemText))

{

this.**setState** ({

todoItems : [...this.state.todoItems,

{action : this.state.newToDoItemText, done: false}], newToDoItemText : ""

});

}

}

**changeStateData** = () => {

this.**setState**(

{userName: this.state.userName === "Stjepan" ? "Dinosaur" : "Stjepan"}

)

}

**render**()

{

return(

<div>

<h4 *className*="bg-primary text-white text-center p-2">

{this.state.userName} Todo List

({this.state.todoItems.**filter**(*t* => !t.done).length}) items to do

</h4>

<div *className* = "container-fluid">

<div *className*="m-1">

<input *className*="form-control" *value*={this.state.newToDoItemText} *onChange*={this.updateNewToDoItemText}></input>

<button *className*="btn btn-danger mt-1" *onClick*={this.createNewToDoTask}>

Add a new task

</button>

</div>

</div>

</div>

);

}

}

### Array of objects into state - explained

constructor(*props*){

*super*(props);

this.state = {

userName : "Stjepan",

todoItems : [

{action:"Buy a flowers", done:false},

{action:"Do workout", done:true},

{action:"Study React", done:false},

{action:"Call a friend", done:true}],

newToDoItemText:" "

}

}

Items of this array are objects which are consisted of two key-value pairs

### Updating state variable

newToDoItemText is empty value which will accept any new value

**updateNewToDoItemText** = (*event*) => {

this.**setState**({newToDoItemText : event.target.value});

}

This method will take any text from text box:

<input *className*="form-control" *value*={this.state.newToDoItemText} *onChange*={this.updateNewToDoItemText}></input>

And put it directly into our state variable, using built-in method setState:

newToDoItemText:" "

### Dynamic update array of objects

**createNewToDoTask** = () =>

{

if (!this.state.todoItems.find(*item* => item.action === this.state.newToDoItemText))

{

this.**setState** ({

todoItems : [...this.state.todoItems,

{action : this.state.newToDoItemText, done: false}], newToDoItemText : ""

});

}

}

First it will check do we already have our todo task, or not:

if (!this.state.todoItems.find(*item* => item.action === this.state.newToDoItemText))

If we don’t have task, it will be added into our toDoItems array:

this.**setState** ({

todoItems : [...this.state.todoItems,

{action : this.state.newToDoItemText, done: false}], newToDoItemText : ""

});

Three dots after [ :

todoItems : [...this.state.todoItems

They are indicating spread operator – which is used for expanding our array in this case.

### Render – explained

Following code will count how many uncompleted tasks we have in toDoItems list:

({this.state.todoItems.**filter**(*t* => !t.done).length}) items to do

It will check boolean values for done keys for each of our toDoItems.