Learn React JS

# **I - Chapter I : JavaScript for React**

## 1) Variable declaring :

• Const: A **const** variable once is declared and initialized cannot reassign a new value to it.

## 2) Variable scoping :

If a variable with type **var** is created inside an ‘if/else’ statement or in a ‘for’ loop for example it will be accessible even from outside its block and will then not protect its global variable. However, a **let** variable will protect the global variable so the variable created inside of a block will only be scoped inside of it and not globally.

## 3) Template Strings :

Traditionally, we used the + sign to concatenate a string by adding variable values to it.

console.log (firstname + “ “ + lastname)

With the template string we can do that by using a $ sign

console.log (‘ ${firstname}, ${lastname) ’)

With template strings, we can directly make a whitespace without adding “ “ because it supports it so we can read and write strings more easily.

const email = Hello ${firstname},

Thanks for ordering our product Mr. ${lastname}.

Thanks.

Mr. ${agent}

## 4) Function expression :

Instead of creating a normal function and call it we can use function expression which assigns a function to a variable.

const Clickme = function() {

console.log(“Clicked”);

};

Clickme();

However, with a function expression we cannot call the function before the function declaration. So, in this example the call of the function is after the declaration so we will have no problem with that.

const ClickToGetName = function(firstname, lastname) {

console.log(‘ My firstname is ${firstname} and my lastname is ${lastname} ‘);

}

ClickToGetName(“jack”, “Charles”);

• Function expression with return type

const ClickToReturn = function(firstname, lastname) {

return ‘ ${firstname} : ${lastname} ‘

}

Console.log(ClickToReturn(“jack”, “Charles”));

• Default parameters

We can set up default parameters for a function

const ClickToReturn = function(firstname = “Jackie”, lastname = “Hanna) {

return ‘ ${firstname} : ${lastname} ‘

}

Console.log(ClickToReturn());

## 5) Arrow Function :

With arrow function we can create a function without mentioning the ‘function’ keyword. Also that is applicable often to the ‘return’ keyword.

• Function without arrow :

const TurnToLord = function (name) {

return ‘Lord of ${name}’

}

• Function with arrow :

const TurnToLord = name => ‘Lord of ${name}’;

So with arrow its is a lot more simple. In this example we have declared only one arugment so it is not necessary to put parentheses around it but if we declared several arguments, parantheses are needed in that case.

const TurnToLord = (name,lastname) => ‘Lord of ${lastname}’;

Here for the return we are directly returning what needed to be returned but if some check must be done before that we will add the ‘return’ keyword and the curly braces so the code will be as following :

const TurnToLord = (firstname, lastname) => {

if (fistname == “Jacob”)

{

return ‘${lastname}’;

}

};

• Returning objects with arrow function :

If we need to return an object using arrow function we just wrap up the object with ( )

const ReturnObject = (firstname, lastname) => ( {

first : firstname,

last : lastname

} );

console.log (ReturnObject(“john”, “cameroon”));

• Arrow function and scope :

a) Normal function without arrow :

Regular funtions do not block **this**.

const objet = {

mountains : [ “Freel”, “Rose”],

print: function(delay: 1000) {

setTimeout (function() {

console.log(this.mountains.join(“,”) );

} delay) ;

}

};

In this case **this** is not protected and it does not refer to mountains but to another object. So in this we can use arrow function which protects this.

const objet = {

mountains : [ “Freel”, “Rose”],

print: function(delay: 1000) {

setTimeout (() => {

console.log(this.mountains.join(“,”) );

} delay) ;

}

};

## 6) Compiling JavaScript :

In the past, when a new feature is added to JavaScript, developers must wait weeks and months so that the feature is added to the browser they would like to use. However, that is not the case In our days where we can use a compiling tool to check that all is working well in our code. Now for JavaScript, **Babel** is one of the most popular compiling tool which made it possible to use latest JavaScript feature right away.

JavaScript has now source code which means that it will have some files that belong to your project and not set on the browser.

## 7) Object and Arrays :

• Destructuring objects : Allows us to locally scope fields in an object so we can choose which values of object to be used.

const sandwish = {

bread : “white bread”,

meat : “tuna”,

cheese : “cheddar”,

toppings : [“lettuce”, “tomato”]

};

const { bread, meat } = sandwish

Here the code will take the value of bread and meat from the object and will create a local varibale for each one, however as that we created bread and meat as **let**, the changing of the values of the local variables will not implies a change of value of the object variables because **let** protects the scoping.

• Destructuring parameters :

In the following code we used person.firstname to get access of the value firstname in person.

const ExtractFromObject = person => {

console.log(‘ ${person.firstname}’);

};

const person = {

firstname : “Zack”,

lastname : “Cole”

};

ExtractFromObject(person);

However, we can use the benefit of destructuring parameters and without the dot notation which will be written like this :

const ExtractFromObject = ( {firstname} ) => {

console.log(‘ ${firstname’) ’) ;

};

Now let us get a little more complicated…

const person = {

firstname : “Zack”,

lastname : “Cole”

spouse: {

firstname: “Jenny”

lastname: “Cole”

}

};

const ExtractFromObject = ( {spouse: {firstname} } ) => {

console.log(‘ ${firstname’);

};

• Destructuring arrays :

const [firstanimal] = [“Horse”, “Monkey”, “Dog”];

console.log (fisrtanimal) // Horse

If we want to access the last element for example in the list we can use commas to pass from an index to another.

const [ , , lastanimal] = [“Horse”, “Monkey”, “Dog”];

console.log (lastanimal) // Dog

• Object literal enhancement : It is the opposite of destructuring. It is the fact of restrucuring object values back together.

const bread = “white”;

const meat = “Tuna”;

const sandwish = { bread, meat };

console.log (sandwish) // { bread : white, meat : Tuna }

We can also add methods to an object with the restructuring method. We just add the name of the method like what we did for the variables bread and meat.

## 8) The spread operator :

It’s the three dots (…) that allows us to do several tasks. First, it allows us to combine the contents of arrays. So let us take an example of combining two arrays in a third new array.

const array1 = [ “horse”, “elephant” ] ;

const array2 = [ “dog”, “cat” ] ;

const array3 = […array1, …array2] ;

console.log(array3.join(,)) ; // horse, elephant, dog,cat

Lets take an example to see where the spread operator can solve a certain problem.

const array1 = [“dog”, “cat”] ;

const [last] = array1.reverse() ;

console.log (last) // cat

console.log (array1.join(,)) // cat, dog

In this example we can see that the array1 has been changed because of the function reverse and that is normal but what if we did not want that to happen in that case we could use the spread operator because it creates a copy of that array.

const array1 = [“dog”, “cat”] ;

const [last] = […array1].reverse() ;

console.log (last) // cat

console.log (array1.join(,)) // dog, cat

The spread operator can also be used to create a copy of an array by getting all the elements of the array without the first element of it.

const array1 = [“dog”, “cat”, “mouse”] ;

const [first, …array2] = array1 ;

console.log (array2) // cat mouse

console.log (first) // dog

The spread operator can also be used as function parameters to a function. In this case the parameters are called rest parameters.

const newFunction = …array1 => {

const [first, …array2] = array1 ;

console.log (‘First town is ${first}’) ;

const [last] = […array1].reverse() ;

console.log (‘Last town is ${last}’) ;

}

newFunction(“Nantes”, “Clisson”, “Getine”);

The spread operator can also be used for objects and the method of using that is the same as for the arrays.

const object1 = {

firstname : “Jay”,

lastname : “Cameroon”

};

const age = 15;

const object2 = {

…object1,

age

};

## 9) Asynchronous JavaScript :

All the things and instructions learned so far are synchronous instructions which means they happen in order. However, in modern web we need to perform asynchronous tasks. With JavaScript, asynchronous tasks do not block the main thread. JavaScript is free to do something else while waiting for an API to return data.

## 10) Simple promises with fetch :

**fetch()** function allows us to make a request to a rest API.

console.log (fetch(“https:/”)) ;

When we log this we see that there is pending promise. The promise is an object that represents wether the async operation is pending, has been completed or failed. So in the fetch result, the pending promise represents a state before the data has been fetched. We need to add a function called **then()** and this function will be executed if the function **fetch()** was successful. So to translate that into a familiar phrase : Fetch some data and then launch a callback function. In the following example we are converting the response into json data.

fetch(“https:/”).then(res => console.log(res.json())) ;

We can add more **then()** functions and each one will take the result of the previous one.

fetch(“https:/”)

.then(res => res.json()) ;

.then(json => json.results) ;

.then(console.log) ;

.catch(console.error) ;

Here we are fetching a get request then the response we will turn it into json then the json data will be put into results and after that we are outputting the results. A catch occurs if the fetch was not done successfully.

## 11) Async / Await :

Another approch to handle promises is to create async functions which looks more familiar as syntax. Instead of using the chain of **then()** function and waiting for the results to be resolved, async function could be told to wait for the promise to resolve before executing any code found in the function. A async function need to have try and catch.

const fakePerson = async() => {

try {

let res = await fetch(“https:”);

let {result} = res.json();

console.log(result);

} catch (error) {

console.log(error);

}

} ;

## 12) Building promises :

When making an asychronous request, we could have many types of successful or unsuccessful requests. Promises gives us a way to simplify back to a simple pass or fail.

const getPeople = count =>

new Promise((resolves, rejects) => {

// put some code

});

getPeople(5)

.then(members => console.log(members) )

.catch(error=>console.error(‘Error’);

Promises make dealing with asynchronous requests easier which is good because we have to deal with a lot of asynchronicity.

## 13) Classes :

Today, React is moving away from classes but using instead functions to construct components but we will definitely still see classes all over the place. React uses something called prototypical inheritance which allows the creation of structures that feel object-oriented.

function vacation (destination, length) {

this.destination = destination;

this.length = length

}

vacation.prototype.print = function() {

console.log(‘Destination ${destination}’) ;

} ;

const malu = new vacation (“malu”, 7);

A vacation has properties (destination, length) and a method print. The malu instance inhertis the print method through the prototype. So functions are objects and inheritance is handled through the protoype.

Now let us take the use of classes not functions :

class Vacation {

constructor(destination, length) {

this.destination = destination;

this.length = length; }

print () {

console.log(‘print destination : ${destination}’) ;

}

}

The class name is normally capitalized.

const object = Vacation(“Chile”, 7);

We can extend classes :

class Trip extends Vacation {

constructor(destination, length, gear) {

super(destination, length);

this.gear = gear;

}

print () {

super.print();

console.log(‘Gear ${this.gear.join(“ and “) )’;

}

const object = Trip (“Malu”, 7, [“sunglasses”, “jeans”]) ;

object.print(); // Gear sunglasses and jeans

## 14) ES6 modules:

It’s a pack of reusable code that can easily be incorporated into other JavaScript files without causing variable collisions. There are two options when creating and exporting modules

* Export multiple JavaScript objects from a single module
* Export one JavaScript object per module

In text-helper.js, 2 functions are exported :

export const print = message => log(message, new Date() )

export const log = (message, timestamp) =>

console.log(‘${message}’);

Export can be used to export any JavaScript type that will be used in other module. Any other variables in text-helper.js other than those who are exported are local to that module.

Modules can also export a single variable and in that case we will use **export default.**

export default new Trip (“Malu”, 7, [“sunglasses”, “jeans”]) ;

So **export default** can be used when we want to export only one type.

To import the exported thing in a module we use the **import** command. Modules with multiple exports can take advantage of object destructuring.

import {print, log} from “./text-helpers”;

import {print as p, log as l} from “./text-helpers”;

We can import everything exported from a module by doing :

import \* as fns from “./text-helpers”;

This import and export syntax is not supported by all browsers and by node but it is supported by Babel.

## 15) CommonJS :

It is the module pattern supported by all versions of node. With CommonJS, JavaScript objects are exported using module.exports

In CommonJS we can export the functions as objects.

const print = message => log(message, new Date() )

export const log = (message, timestamp) =>

console.log(‘${message}’);

module.exports = {print, log} ;

In CommonJS, import is replaced with require

# **II - Chapter II : Functional programming with React :**

## 1) Functional programming (Declarative programming) :

In JavaScripts, functions are variables and since functions are variables, we can add them to objects.

Creating a function as **const** prevent it from being overwritten.

We can also add functions to array in JavaScript.

const array = [“they can be inserted into array”, message => console.log(message) ];

Functions can be sent to other functions as parameters.

const func = logger => {

console.log(logger);

}

func (message => console.log(message) );

We can also return a function like a variable

## 2) Imperative programming vs declarative programming :

Imperative programming requires a lot of comments for the code to understand it.

React is declarative.

• Construct a DOM declarativley using a React component :

const {render} = ReactDOM;

const Welcome = () => (

<div id=”Welcome”>

<h1>Hello World</h1>

</div>

);

render(<Welcome />, document.getElementById(“target”));

Here the welcome component describes the DOM that should be rendered by giving it the ID of the element “target”.

The render function will build the DOM abstracting the details of how the DOM will be rendered so yes React is declarative.

## 3) Functional concepts :

**a) Immutability :** In JavaScript, function arguments are references to the actual data so if I have an object with properties and I created a function to get access to a property of the object and change the value, the value of the object will be changed let’s take the example below :

Example 1:

let object1 = {

title = “rookie”,

rate = 8

};

const func = (obj , rating) => {

obj.rate = rating;

}

func(object1, 5);

console.log(object1.rate)// 5

Now let us see how we can make that in a way so that we not modify the original object and make a copy of it however. (Do not forget the parentheses because we are returning an object for an arrow function)

const func = (obj , rating) => ( {

return Object.assign ( {}, obj, {rate : rating} );

} );

console.log (func(object1, 5).rate) // 5

console.log (object1.rate) // 8

Here we used **Object.assign** which is the copy machine which takes a blank object.

We can also use the spread operator here :

const func = (obj , rating) => ( {

…obj,

rating

} );

Example 2:

Now let us take a list :

var list = [{title : “hi”}, {title : “hello”}]

const addToList = (list, value) => {

list.push ( {value : value} );

return list;

};

console.log (list.length) // 3

So here, the function **push()** is not an immutable function we can use **conact()** instead.

To remove an element of the list we can use **filter()** function instead of **pop()** or **splice()** because **filter()** is an immutable function.

Another array function that is essential to functional programming is **map()** function. It takes a function as a parameter.The function will be invoked once for every item.

const schools = [“Nantes”, “Dijon”]

const highSchool = schools.map (school => ‘${school} High School’);

console.log(highSchool.join(“\n”));

// Nantes High School

// Dijon High School

**map()** function is immutable so schools array has not been changed. In this example we produced an array of string from an array of string. The **map()** function can produce array objects, values, arrays, other functions --- any JavaScript type. Here is an example of the **map()** function returing an object for every school.

const arrayOfObject = schools.map ( school => ( { name : school} ) );

Let’s create a function that changes a property of an object in a list of objects. We can use **map()** for this also.

const schools = [{name : Nantes}, {name : Dijon}];

const function = (oldName, newName, array) => {

array.map(school => {

if (school.name === oldname) {

return {

… school,

newName

};

} else {

return school;

} });

So here we are creating a new array of objects based upon the original array. We can also write it in one line.

const func = (oldName, newName, array) => array.map( item => (item.name === oldname ? {…item, newName} : item) );

• If we want to transform an array into an object we can use **map()** function with **Object.keys** where **Object.keys** is a method that can be used to return an array of keys from an object.

Let us take the example of transforming the schools object into an array of schools.

const schools = {

Dijon : 10,

Paris : 15

};

const schoolArray = Object.keys(schools).map(key => ({name : key, distance : schools[key] }) );

console.log(schoolArray); // [ {name : Dijon, distance : 10}, {name : Paris, distance : 15} ]

So here **Object.keys** returns an array of schools names.

• Now we must also learn the functions **reduce** and **reduceRight**. These functions can be used to transform an array into any value like number, string, object…

Let’s say we need to find the maximum number in an array of numbers.

const ages = [21, 18, 42, 30] ;

const maxAge = ages.reduce( (max, age) => {

console.log(‘${age}>${max} = ${age>max}’);

if (age > max) {

return age;

}

else {

return max;

}

}, 0);

console.log(“MaxAge”,maxAge);

// 21 > 0 = true

18 > 21 = false

42 > 21 = true

30 > 42 = false

MaxAge 42

So here the reduce function takes two parameters: A callback function and an original value. In this case the original value is 0 which sets the initial maximum value to max.

We can do that in one line:

const maxAge = ages.reduce( (max, age) => ( age>max ? age : max), 0 );

Array.reduceRight works the same as Array.reduce but the difference is that it starts reducing from the end of the array rather from the beginning.

**b) Pure functions:** A function that returns a value that is computed based on its arguments. It return always a value or another function and it does not affect the actual object sent to it. Pure functions are another core concept of functional programming. they make our life easier and without side effects (without changing our application’s state). When writing functions, we will try to follow these 3 rules:

- The function should take at least one argument.

- The function should return a value or another function.

- The function should not change or mutate any of its arguments.

**c) Higher-order functions:** They are functions that can manipulate other functions. They can take functions in as arguments or return functions or both. **Array.map**, **Array.filter** and **Array.reduce** all take functions as parameters. Let us see how we can implement a higher-order function.

**Currying** is a functional technique that involves the use of higher-order functions.

const userLogs = username => message => console.log(‘${userName} -> ${message}’) ;

const log = userLogs (“georges”);

log(“Attempted to load 20 fake numbers”);

getFakeMembers(20).then(

members => log (“successful”);

error => log (“Error”);

);

// georges -> Attempted to load 20 fake numbers

georges -> successful

georges -> Attempted to load 20 fake numbers

georges -> Error

**d) Composition:** We can chain together multiple functions

const template = “hh:mm:dd tt”;

const clockTime = template

.replace(“hh”, “03”)

.replace(“mm”, “33”)

.replace(“ss”, “44”)

.replace(“tt”, “PM”);

console.log (template); // “03:33:44 PM”

Here the function replace is immutable so the template will not change.

• **Both()** function : It is one function that pipes a value through 2 separate functions.

const both = date => appendAMPM (civilianHours(date) );

The output of civilianHours become the input to appendAMPM

# **III - Chapter III : How React works :**

With React, we will be building our applications with JSX. JSX is a tag-based JavaScript syntax that looks a lot like HTML. To understand React, we must understand React elements, React components and see how we can create custom components that compose other components and elements.

## 1) Requirements :

In order to let React work in the browser, we need to include 2 libraries : React and ReactDOM. React is the library for creating views. ReactDOM is the library used to actually render the UI in the browser. Both libraries are available as scripts from the unpkg CDN

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

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

We will be using the development version of React so we can see all the error messages and warnings in the browser console.

## 2) React elements :

The elements that make up an HTML document become DOM elements when the browser loads HTML and renders the user interface. In the past, websites consisted of independent HTML documents. When the user navigated these pages, the browser would request and load different HTML documents. The invention of AJAX (Asynchronous JavaScript and XML) brought us the single-page application or SPA. Since browsers could request and load tiny bits of data using AJAX, entire web application could now run out of a single page and rely on JavaScript to update the user interface. In the SPA, the browser loads initially one HTML document. As user navigates through the site, he actually stays on the same page. JavaScripts destroys and creates a new user interface as the user interacts with the application. It may feel like jumping from one page to another but actually the user is still on the same HTML page and JavaScript is doing the heavy lifting.

The DOM API is a collection of objects that can be used by JavaScript to interact with the browser to modify the DOM. Everytime we use **document.createElement** or **document.appendChild** we are actually working with the DOM API.

React is a library that is used to update the browser DOM for us. With React we do not interact with the DOM API directly. Instead, we provide instructions for what we want React to build and React will take care of rendering the elements we want to create.

The browser DOM is made up of DOM elements. Similarly, the React DOM is made up of DOM elements. DOM elements and React elements may look the same but they are actually quite different. A react element is a description of what the actual DOM element should look like. We can also say that React elements are the instructions for how the browser DOM should be created.

We can create a React element to represent an h1 using **React.createElement**

React.createElement (“h1”, { id : “recipe-0”}, “Baked Potatos”);

The first argument is the type of element we want to create. The second argument represents the element’s properties. The third argument represents the element’s children properites (any nodes that are inserted between the opening and closing tag, in this case we put some text)

During rendering, React will convert it to actual DOM.

<h1 id = “recipe-0”>Baked Potatos</h1>

And the strucure of the React element will look like this:

{

$$typeof: Symbol (React.element);

“type” : “h1”,

“key” : null,

“ref” : null,

“props” : {id: “recipe-0”, children : “Baked Potatos”},

“\_owner” : null,

“\_store” : {}

}

There are fields that are used by React: **\_owner, \_store, $$typeof.**

The key and ref are important to React elements but we will introduce those later. For now we will take a close look at the **type** and **props** fields. The **props** property represents the data and child elements required to construct a DOM element. The children property is for displaying other nested elements as text.

## 3) ReactDOM :

After the creation of a React element, we need to render it to the browser to make it appear there. ReactDOM contains the tools that are needed to render React elements in the browser. In ReactDOM we will find the **render** method. We can render a React element, including its children, to the DOM with **ReactDOM.render.** The element we want to render is passed as the first argument and the second argument is the target node where we should render the element.

const dish = React.createElement (“h1”, null, “Baked Salmon”);

ReactDOM.render (dish, document.getElementById (“root”) );

This will be converted like this:

<body>

<div id=”root”>

<h1>Baked Salmon</h1>

</div>

</body>

So anything concerning rendering elements is found in the ReactDOM package. Now with the version 16+ with React, we can render an array and not just one element.

const dish = React.createElement (“h1”, null, “Baked Salmon”);

const dessert = React.createElement (“h2”, null, “Baked Potatos”);

ReactDOM.render([dish,dessert], document.getElementById(“root”) );

## 4) Children :

React use **props.children** to render children elements. We rendered in the previous section a children which is a text element for the header h1 so the **props.children** was equal to ‘’Baked Potatoes” for example. We can make a tree of elements by rendering other React children. So, we will have a tree which a root and as much branches as we want. Let us take an example of an unordered list and render it with React.

<ul>

<li> “Pizza”</li>

<li> “Pasta”</li>

<li> “Hamburger”</li>

</ul>

Now we will use React to render such an element.

React.createElement (“ul”, null, React.createElement (“li”, null, “Pizza”),

React.createElement (“li”, null, “Pasta”),

React.createElement (“li”, null, “Hamburger”)

);

Every additional argument sent to the method is a children element. React creates an array containing these children and set the value of **props.children** to that array.

## 5) Classname in React :

**class** is a reserved word in JavaScript so we need to use **classname** to define the class property of an HTML element. Any element that has an HTML class attribute is using **classname** for that property instead of **class.**

## 6) Constructing elements with data :

So in the previous example where we rendered the unordered list we was rendering some text. In fact, since React is a JavaScript, we can use JavaScript for our favor. We will create an array containing the text of each list element and then we will use the **map()** function to render it.

const list = [“Pizza”, “Pasta”, “Hamburger”];

React.createElement (“ul”, { classname : “food”},

list.map(item => React.createElement (“li”, null, item) ) );

However, when we run this code, we will have a warning saying that each child in an array should have a unique “key” prop.

When building a list of child elements by iterating through an array, React prefers that each child have his own key property which helps him updating the DOM efficiently. So, we can make this warning disappear by adding a unique key property to each element. We can use the array index for each element as the key.

const list = [“Pizza”, “Pasta”, “Hamburger”];

React.createElement (“ul”, { classname : “food”},

list.map( (item, i) => React.createElement (“li”, {key : i}, item) )

);

## 7) React components :

UI are composed of many parts (Headers, buttons list…). So, if we take the same component for 3 different examples or data, we will have the same elements in the component but each example has its own value for these elements. In React, we describe each of these parts as a component.

We will create a component by writing a function. That function will return a reusable part of the UI. Let us create a function that returns an unordered list of food.

const list = [“Pizza”, “Pasta”, “Hamburger”];

function FoodList () {

return React.createElement (“ul”, { classname : “food”},

list.map( (item, i) => React.createElement (“li”, {key : i}, item) )

); }

ReactDOM.render (React.createElement (FoodList, null, null), document.getElementById(“root”));

The component name is FoodList and the function output elements that look like this:

<FoodList>

<ul classname = “food”>

<li key = “0”>Pizza</li>

<li key = “1”>Pasta</li>

<li key = “2”>Hamburger</li>

</ul>

</FoodList>

But what if we create a component and then we pass the data to it as properties so that we could use as many arrays as we want dynamically.

function FoodList () {

return React.createElement (“ul”, { classname : “food”},

array.map( (item, i) => React.createElement (“li”, {key : i}, item) )

); }

ReactDOM.render (React.createElement (FoodList, {array : list}, null), document.getElementById(“root”));

<FoodList array =”[“Pizza”, “Pasta”, “Hamburger”]”>

<ul classname = “food”>

<li key = “0”>Pizza</li>

<li key = “1”>Pasta</li>

<li key = “2”>Hamburger</li>

</ul>

</FoodList>

So here the component property is an array that contains the food. Another adjustment we can make here is to access the array to map through the props

function FoodList () {

return React.createElement (“ul”, { classname : “food”},

props.array.map( (item, i) => React.createElement (“li”, {key : i}, item) )

); }

We could also clean up the code a bit by destructuring ‘array’ from props.

function FoodList ({array}) {

return React.createElement (“ul”, { classname : “food”},

props.array.map( (item, i) => React.createElement (“li”, {key : i}, item) )

); }

## 8) Older React :

Before the newest versions of React, components were created in a different way where the use of **React.createClass()** method was needed. Components that used **React.createClass()** would have a **render()** method that described the React element to be rendered. After that and after those classes were added to JavaScript with ES 2015, React introduced a new method for creating components and that was by using class syntax.

class FoodList extend React.Component {

render() {

return React.createElement (“ul”, { classname : “food”},

props.array.map( (item, i) => React.createElement (“li”, {key : i}, item) ) )

}

}

**Now we use functions to create components but it is still supported with the use of class.**

# **IV - Chapter IV : React with JSX :**

React developers do not use the **createElement()** function and make complex barely readable trees of JavaScript. Developers use JSX which is the mixture of JavaScript and XML. It is a JavaScript extension that allows us to define React elements using a tag-based syntax directly within the JavaScript code. Sometimes JSX is confused with HTML because they look similar. JSX is a way of creating React elements which is a simpler way than using the **createElement()** function.

## 1) React elements as JSX :

Facebook’s React team released JSX with React to make React more readable and look like HTML and XML. In JSX, an element type is specified with a tag. The tag’s attribute represent the properties. The element’s children can be added between the opening and the closing tag. We can also add other JSX elements as children. JSX works also with components bu simply defining the component using the class name.

Without JSX: React.createElement(FoodList, {array : list}, null);

With JSX: <FoodList array = {list} />

So when passing JavaScript values to components as properties we need to surround it with curly braces. Component properties take 2 types: A string or a JavaScript expression which can include arrays, objects and even functions and to include them we must surround them with curly braces.

## 2) JSX tips :

JSX looks familiar but there is some considerations that we should understand when working with JSX.

**a) Nested components:** JSX allows us to add a component as children for another component. For example, inside of the **FoodList** component we can add many other components inside of it.

<FoodList>

<AnotherList />

<AnotherList />

</FoodList>

**b) className :** Since **class** is a reserved word with JavaScript, **className** is used to define the class attribute of an element.

<h1 className = “fancy”>Baked Potatos</h1>

**c) JavaScript expressions:** They are covered with curly braces and indicate where variables will be evaluated and their resulting values returned. For example, if we want to display the value of the title property in an element, we can insert that value using a JavaScript expression, the variable will be evaluated and its value returned.

<h1>{title}</h1>

Values of types other than String should also appear as JavaScript expression:

<input type = “checkbox” defaultChecked = {false} />

**d) Evaluation:** The JavaScript added between curly braces will be evaluated which means that any operation inside of curly will occur by JavaScript.

<h1> {“Hello” + title} </h1>

<h1> {title.toLowerCase().replace} </h1>

## 3) Mapping arrays with JSX :

JSX is JavaScript so we can incorporate JSX directly inside of JavaScript functions. For example, we can map an array to a JSX element:

<ul>

{props.List.map ( (item, i) => (

<li key = “{i}> {item} </li> ) )

}

</ul>

JSX looks clear and readable but it needs **Babel** to convert the code into **createElement** calls and so that it can be interpreted with a browser.

## 4) Babel :

JavaScript is an interpreted language: The browser interprets the code as text, there is no need to compile JavaScript. However, no browser supports JSX and not all browsers support the latest JavaScript syntax. So, since we are using JSX and the latest JavaScript versions, we need a compiler to convert our fancy code and that’s what **babel** is designed for. The first version of the project was called 6to5, released in September 2014 which is a tool that was used to convert ES6 to ES5 that is supported by all web browsers. And now it supports converting JSX into JavaScript. Babel is used in production at Facebook, Netflix, PayPal… Previously, Facebook had created a JSX transformer that was their standard but it was then replaced by Babel.

The easiet way to work with babel is by including a link to the babel CDN directly in our HTML which will compile any code in script blocks that have a type of “text/babel”. Babel will compile the source code on the client before running it. It is not the best solution for production but it is a great way to get started with JSX.

<script type=”text/babel”> //some code here or link to separate JavaScript file having JSX</script>

## 5) Recipes as JSX :

let’s consider we have the following array containing two objects :

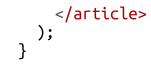


Each object contains the name of the recipe, a list of the ingredients required and the steps.

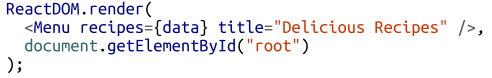
We can create a UI for these recipes with two components : A Menu component for listing the recipes and a Recipe component that describes the UI for each recipe.

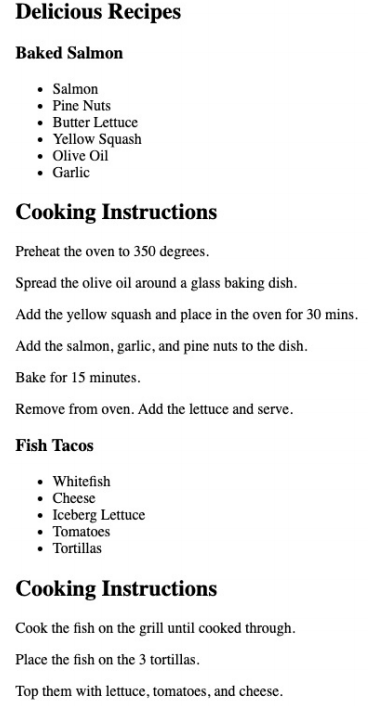
We will first render our Menu component to the DOM and we will pass our data to this component as a property called recipes.





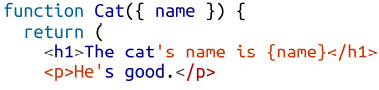






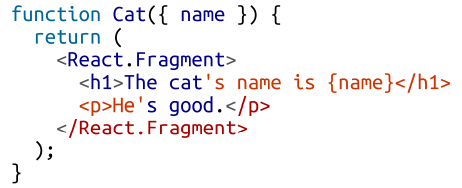
## 6) React fragments :

In the previous section, we rendered the menu component which is a parent component that renders the Recipe component. We will take a moment to look at a small example of rendering 2 sibling components using a React fragment.





In this case we will directly have an error saying that Adjacent JSX elements must be wrapped in an enclosing tag and recommends using a fragment. This is where the role of fragments comes to place. React will not render two or more adjacent or sibling elements as component so we need to have to wrap these in an enclosing tag like a div. This led to a lot of unnecessary tags being created and a bunch of wrappers without much purpose. If we use a React fragment, we can mimic the behavior of a wrapper without actually creating a new tag.



We can also use <> and </> instead of <React.Fragment> and </Reat.Fragment>.

If we take a look to the DOM the fragment will not be shown in the resulting tree. Fragments are a relatively new feature to React which get rid of additional unuseful tags that can pollute the DOM.

## 7) Intro to webpack :

Webpack emerged to be one of the leading tools for real projects. The React ecosystem includes tools like create-react-app, Gatsby and Code Sandbox. When using these tools all code abstraction behind them is hidden. We can up our own webpack build but it not much important for the present time. Webpack is billed as a module bundler which takes all of our different files (JavaScript, CSS, JSX…) and turn them into a single file which brings 2 main benefits: Modularity and network performance. Modularity will allow us to break down our source code into parts or modules that are easier to work with especially in a team environment. Network performance is gained by having to load one dependency in the browser: the bundle

## 8) Creating the project :

We will begin by creating a folder called recipes-app. And for the project we are going to go through the following steps:

a- Create the project

b- Break down the recipe app into components that live in different files.

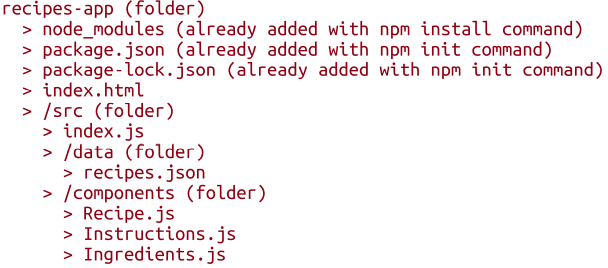
c- Set up a webpack build that incorporates Babel.

There is a tool called **Create React App** that can be used to autogenerate a React Project with all of this preconfigured

**a) Create the project:** We will now create the project and package.json file with npm by sending the -y flag to use all of the defaults parameters. We will install also webpack, webpack cli, react and react-dom.



After that we will create the following directory structure for the components.



**b) Break components into modules:** It is better to create multiple components instead of one big component doing everything because in that way it will be easy for developer to optimize and make code changes when everything is well organized and separated.

**c) Creating the webpack build**

**npx create-react-app “my-app”**

This cli command auto generates our project which let us use directly React without the manual configuration of webpack, Babel, ESLint and associated tools. The use of **npx** let us run **create-react-app** without globally installing it

# **V - Chapter V : React state management :**

In the last chapter we used components and properties to flow user data. In fact, properties are half of the picture. State is the other half. The state of a React application is driven by data that has the ability to change. Introducing state to the recipe application that we created before will make it possible for chefs to create new recipes, modify existing recipes and remove old ones. There is a relationship between state and properties. In this chapter we will introduce state to our application. We will learn to create stateful components and how state can be sent down a component tree and UI back up the component tree. We will learn techniques for collecting form data from users and we will introduce stateful context providers so that we can separate concerns within our app.

## 1) Building a Star Rating component :

The star rating component is one of the most important React components to build.



To begin, we need a star icon and that can be done by getting one from npm library.

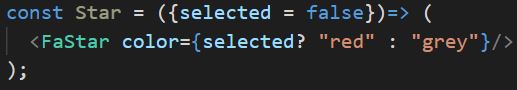
npm i react-icons

react-icons is an npm library containing hundreds of SVG icons that are distributed as React components.



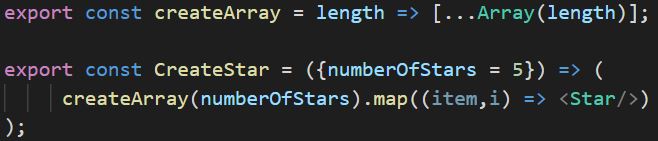
Here, a StarRating component rendering 5 SVG stars.

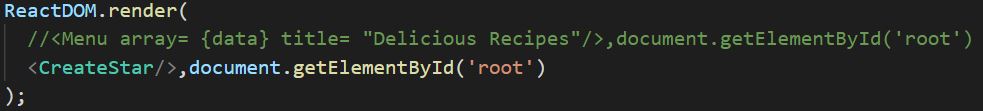
Let’s now create a component that automatically files the stars upon the selected property.



So here, the Star component renders a Star which has a color based on the property it took. If no value was passed it will take a false as default and then renders a grey star and if the property passed which is **selected** was true then a red star will be rendered.

Now we will create a component and a function that takes as parameter the number of stars wanted to be used for the Star-Rating process and we will create these stars based on the number of stars wanted to be used.







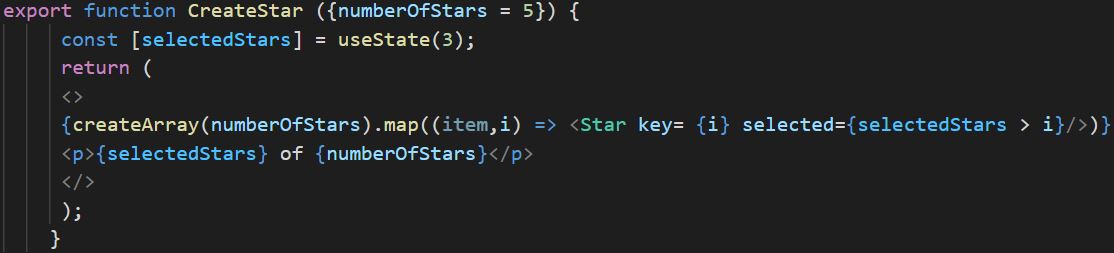
By did not give a value for the property of **CreateStar** when rendering so it took its default value which is 5 and create an empty array of that length and it will pass this array where each time it will create a grey star.

## 2) The useState Hook :

In this part, we will make the Star Rating component clickable which will allow the user the change the rating. Since the rating is a value that will change, we will store and change the value using React state. We incorporate state into functional component using a React feature called **HOOKS** which contains reusable code logic separate than the component tree. They allow us to hook up functionality into our components. In fact, there are many out of the box built in HOOKS that we can use. So, we want to add state to our React component and to do that we will use the **useState Hook.** We simply need to import it from the React library.

Import {useState} from “react”;

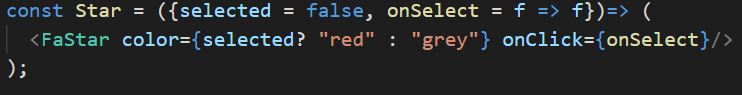
We will create a state variable called **selectedStars** that will have the user rating. This variable will be created by adding the **useState** hook directly to the StarRating component.



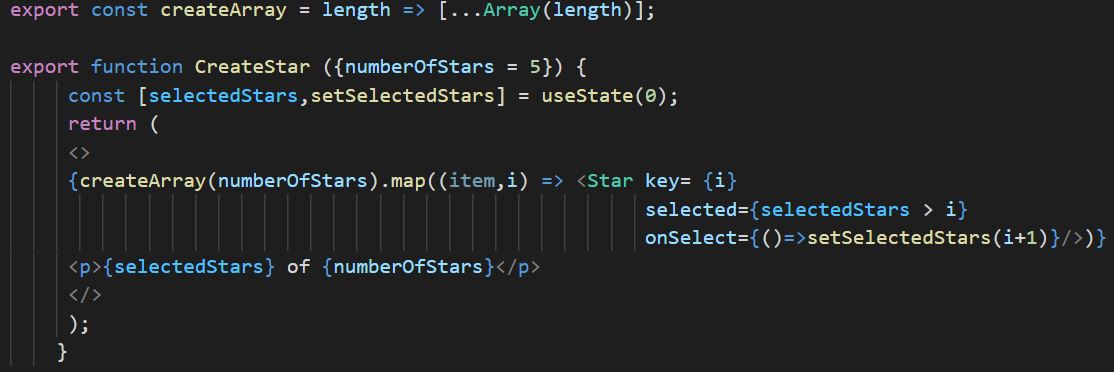
So here we hooked the component with the state. The **useState** hook is a function that we can invoke to return an array and the first value of the array is the state variable that we want to use. The value we send to the useState function is the default value for the state variable. So in this case, **selectedStars** will be of value 3.



Now we want to make the stars clickable and this is done by adding an **onClick** handler to the **FaStar** component.



So now when the user clicks on a star we will invoke the **onSelect()** function. The default value for this function is f=>f. This is simply a fake function that does nothing, it just return whatever argument was sent to it, however we must do that because if we not define the function onselect() an error will occur and here we need a function because the value of onClick is a function. Now that our star is clickable we will use it to change the state of the star rating.



So here, we need to change the value of the state when the user clicks the star. The other item added to the array is a function which is used to change the value of the state. When the user clicks a star we will increment the **selectedStars** value and the component will be rendered which will change the color of every star to the red color because the condition **selectedStars > i** for some cases depending on which star we clicked.

## 3) The old way of React state :

In previous versions of React (v16.8.0 and earlier), the only way to add a state for a component was with the use of a class component which requires more code and complication. We will now convert our functional component **CreateStar** to a class component.

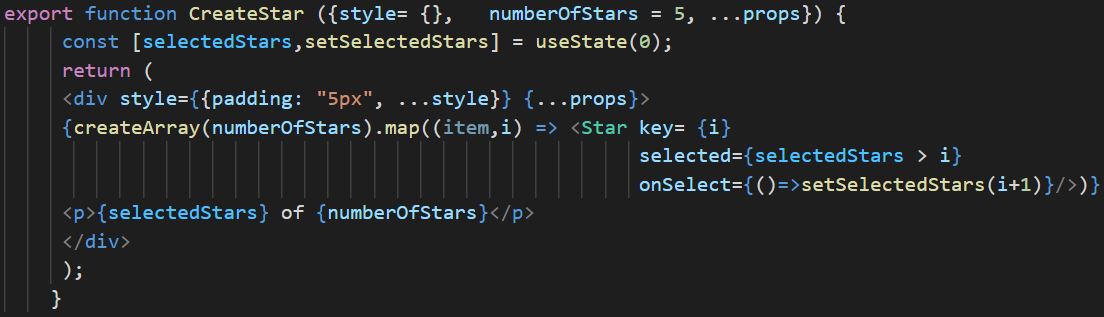
We need first of all to import **{component}** from React.

Import {component} from “react”;



## 4) Refactoring for advanced reusability :

So now, our star rating component can be used across our several applications. However, we need to handle more use cases if we want to ship this component to npm. First let us consider the **style** property. This property allows us to use CSS styles to elements. All react elements have style properties as well lot of components. For our **CreateStar** component we should replace the React fragment with a div so that we can style our component.



So here, we are adding the **style** property for our component. By default, we give it a padding and the rest of the style properties will be passed as argument to the component. Other than that, we are also adding the **props** so that we support the consumer of our component and we assume if he’s going to add some other properties to our component.

## 5) State in component trees:

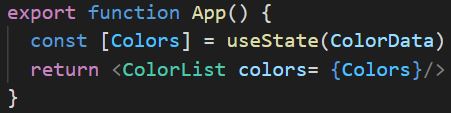
It is not recommended the excess use of states in our components because it will be harder to track problems and bugs. This occurs because it is hard to keep track of where the state values live within our component tree. A solution for that is to store state at the root of the component tree and passing it down to child components via props. We will build a small application that can be used to save a list of colors and it will allow users to associate a list of colors with a custom title and rating. We will use the following dataset:

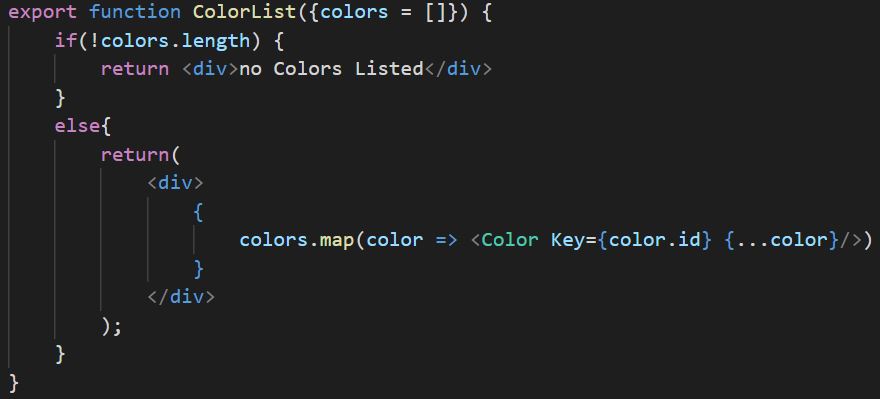


The **color-data.json** file contains an array of three colors. First, we will be creating a UI consisting of React components that will be used to display this data in a browser. After that, we will allow the users to add new colors and rate and remove colors from the list.

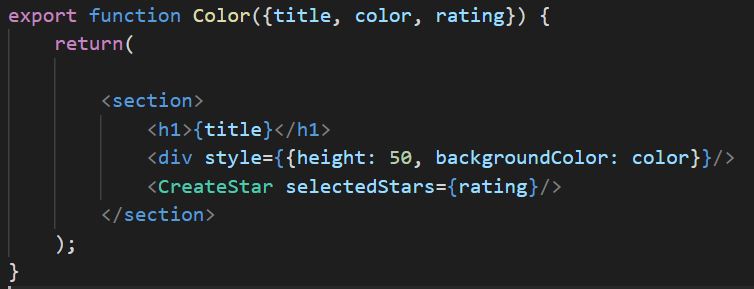
## 6) Sending data down a component tree :

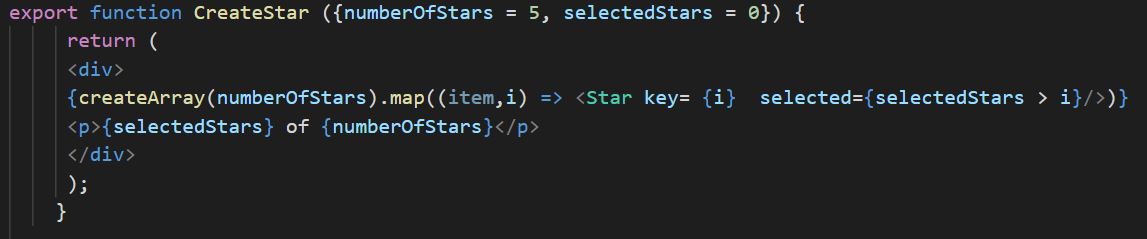
We will store state in the root of the **app** component which is a component within our application that holds state and we will pass the colors to a child component. We will add the list of colors to the app with the **useState** hook.





The **ColorList** receives the colors from the **app** component as props. If the list is not empty, we will render each color item.





So here we modified the **CreateStar** component and it is now a pure component where a pure component does not contain state. We made this component pure because the state for color rating are stored in the **colors** array at the root of our component tree. The goal here is to store state in a single location and not have it distributed through many different components within the tree.

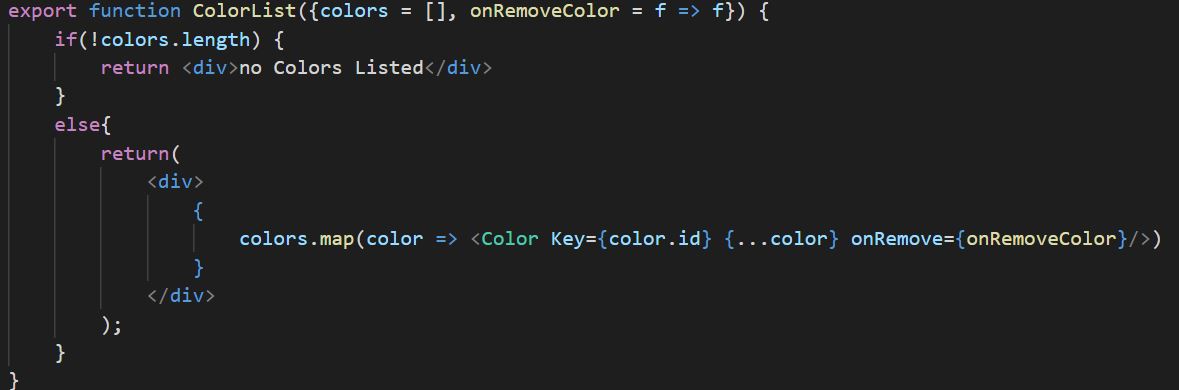


## 7) Sending interactions back up a component tree :

What happens now if we want to remove a color from the list or change the rating of a color in our list? We will need to collect interactions from child components and send them back up the tree to the root component where we can change the state. Let us begin by adding a remove button next each color.



So here, we added a button next to the color which invokes the method **onRemove** by giving it as parameter the id. This function will then be invoked by the parent element and will then invoke a function named **onRemoveColor** which will also have the id passed to it. After that, this function will then invoke the method in the parent element which is the root and it is where we are storing our state. Here we will use the id that was passed as parameter to remove the element from the json list.





So this solution is great because here we can keep track of our state value more simply and we keep our components as a pure component unless the root component that contains the state. Each child component has the role of notifying its parent and passing the info needed to know which item wants the user to remove which is the id in our case till it reaches the root component that contains the state. So when we are changing the state array after the removing of an item all the components will be rerendered.