[React Tutorial 3](#_Toc507196643)

[1 Basic Introduction 3](#_Toc507196644)

[1.1 Let and Const keywords 3](#_Toc507196645)

[1.2 Arrow functions 4](#_Toc507196646)

[1.3 Imports and Exports 4](#_Toc507196647)

[1.4 Classes 5](#_Toc507196648)

[1.5 Spread and Rest 7](#_Toc507196649)

[1.6 Destructuring 8](#_Toc507196650)

[1.7 Reference and primitive types 8](#_Toc507196651)

[2 Project Set up 9](#_Toc507196652)

[2.1 Using React Create-app 9](#_Toc507196653)

[2.2 First look at created project 9](#_Toc507196654)

[2.3 Dynamic content output 10](#_Toc507196655)

[2.4 Dynamic properties 10](#_Toc507196656)

[2.5 Using state inside component 11](#_Toc507196657)

[2.6 Activating the handler and setting state 11](#_Toc507196658)

[2.7 Transferring handler between components 12](#_Toc507196659)

[2.8 Passing params to the handler 12](#_Toc507196660)

[2.9 Two way binding 12](#_Toc507196661)

[2.10 Styling 13](#_Toc507196662)

[3 List and conditionals 13](#_Toc507196663)

[3.1 Conditioning 13](#_Toc507196664)

[3.2 Lists 15](#_Toc507196665)

[3.3 Keys in the list rendering 16](#_Toc507196666)

[3.4 Flexible lists 16](#_Toc507196667)

[4 Styling components 17](#_Toc507196668)

[4.1 Dynamically styling components 17](#_Toc507196669)

[4.2 Dynamically set classes to style components 18](#_Toc507196670)

[4.3 Using Radium library to create :hover effect on style 19](#_Toc507196671)

[4.4 CSS Modules 19](#_Toc507196672)

[5 Debugging react apps 20](#_Toc507196673)

[5.1 Debugging part and insights 20](#_Toc507196674)

[6 Components deep dive 20](#_Toc507196675)

[6.1 Structuring components in better order 20](#_Toc507196676)

[6.2 Splitting into smaller components 21](#_Toc507196677)

[6.3 Stateful vs Stateless components 22](#_Toc507196678)

[6.4 Lifecycle in react 23](#_Toc507196679)

[6.5 Aux implementation (Higher order component – hoc) 24](#_Toc507196680)

[6.6 SetState method properly 25](#_Toc507196681)

[6.7 Wrapping component with hoc 26](#_Toc507196682)

[7 Project 27](#_Toc507196683)

[7.1 Basic setup 27](#_Toc507196684)

[7.2 BurgerBuilder container 28](#_Toc507196685)

[7.3 Prop type validation 30](#_Toc507196686)

[7.4 Burger component 31](#_Toc507196687)

[7.5 Introducing state with array transformation 32](#_Toc507196688)

[7.6 Using reduce method on array 33](#_Toc507196689)

[7.7 Creating build controls 34](#_Toc507196690)

[7.8 Add and remove ingredients 35](#_Toc507196691)

[7.9 Showing the total price 37](#_Toc507196692)

[7.10 Purchasable button and logic for disabling it 38](#_Toc507196693)

[7.11 Adding modal window 39](#_Toc507196694)

[7.12 Showing Modal window 41](#_Toc507196695)

[7.13 Backdrop component 42](#_Toc507196696)

[7.14 Creating buttons for the modal (Order summary) 44](#_Toc507196697)

[7.15 Toolbar 45](#_Toc507196698)

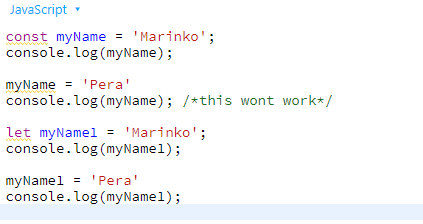
[7.16 Adding the LOGO 46](#_Toc507196699)

# React Tutorial

# Basic Introduction

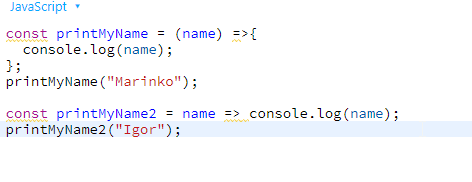
## Let and Const keywords

Instead of var we should always use let or const keywords. Let is used for variable that could be changed further in a code but const cannot, it is constant.



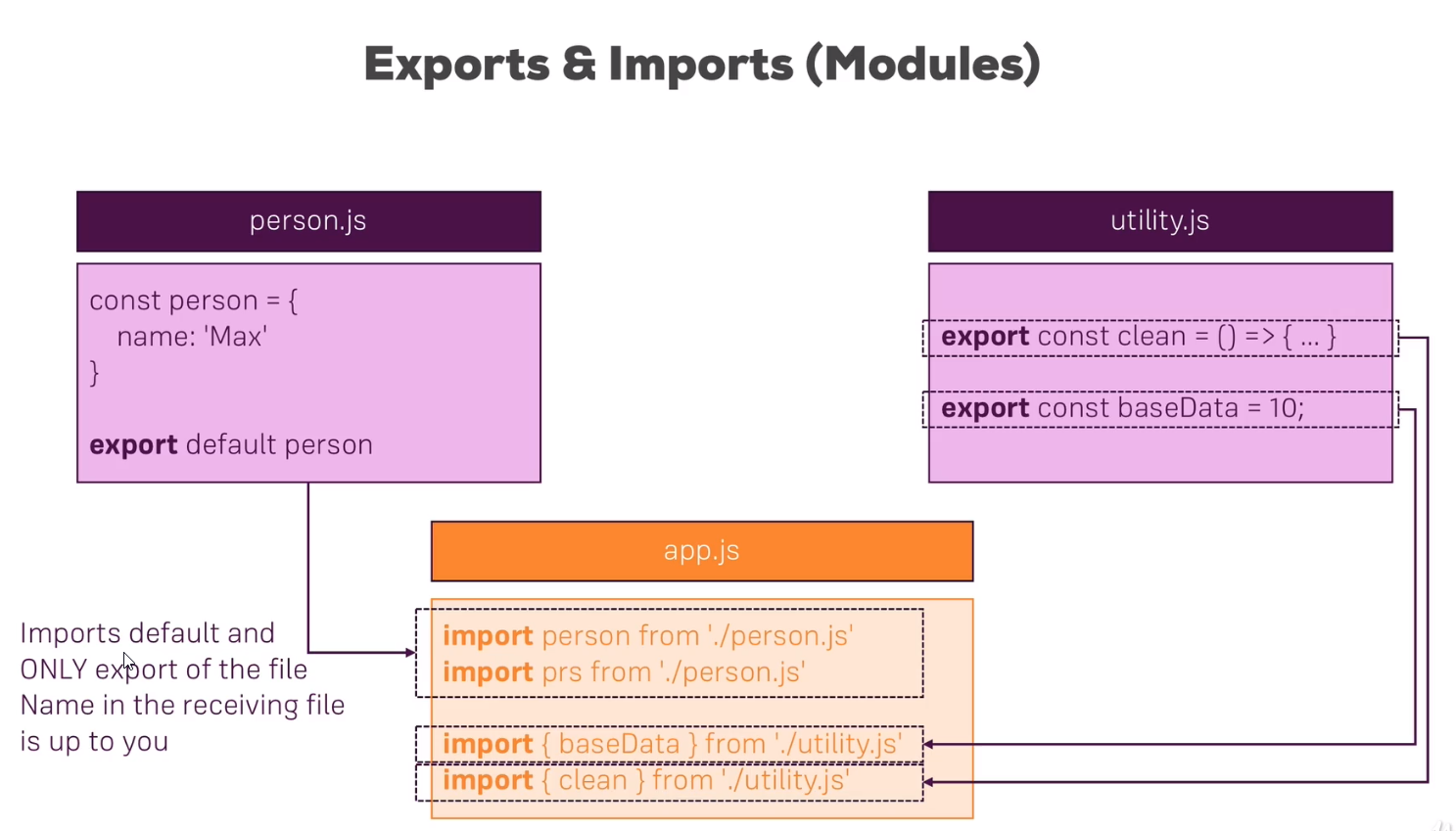
## Arrow functions

We can use normal functions as we used to in java script, but now it is popular to use arrow functions.

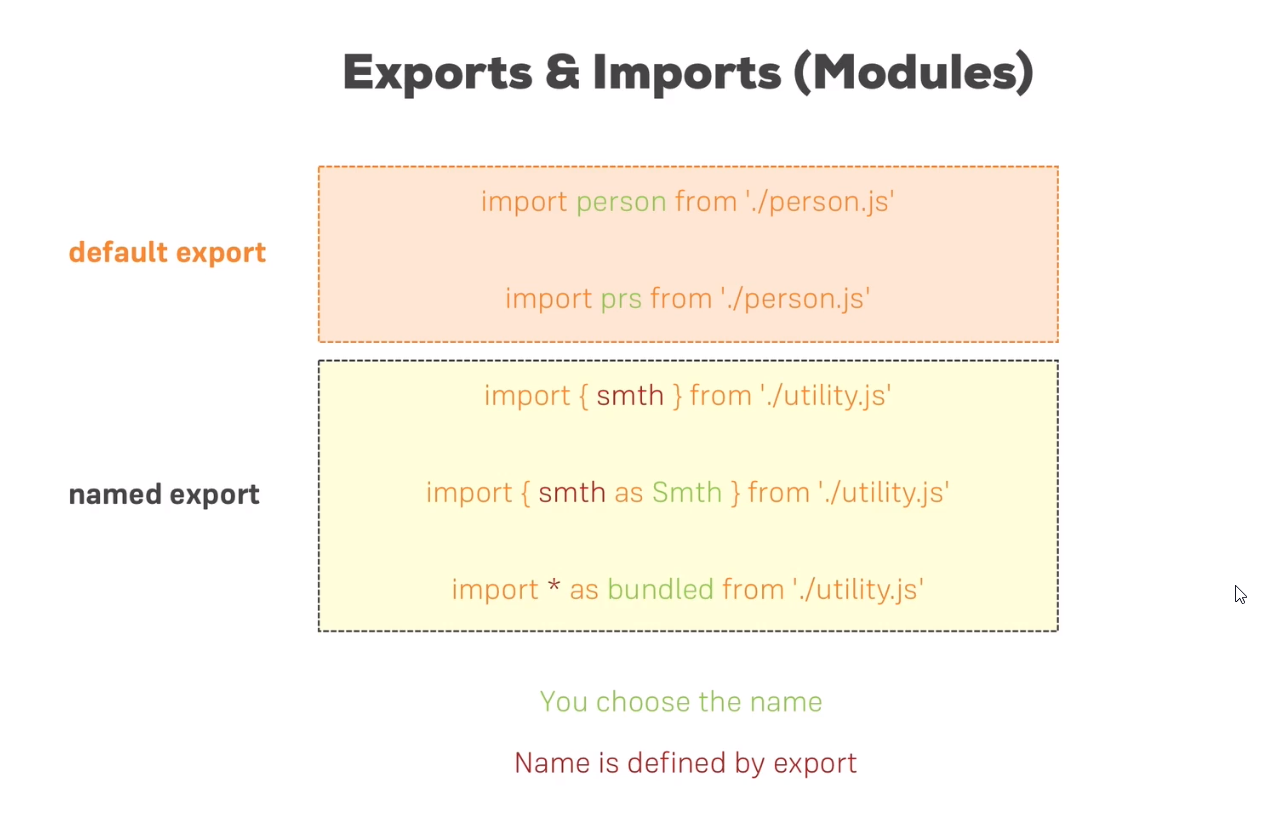


## Imports and Exports

We can export some variables from one file and import it into another file. Two keywords: default and named. With default keyword you can import that variable with any name you want. With named export you can import this variable but only with the same name (between curly brackets) as the name of variable in export file.

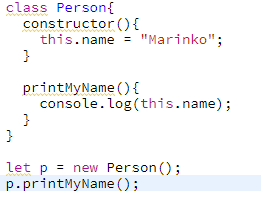


When importing as named, you can add alias to the imported variable with keyword “as”. Or if you importing multiple values, you can wrap them into one object with the \*. And then call them as properties: bundled.something …

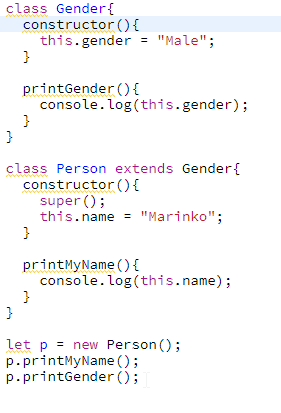


## Classes

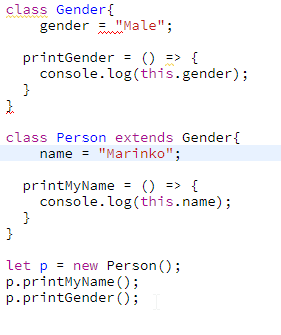
Classes are the same as everywhere.



Classes can inherit from other class with keyword extends. You must use super keyword to call constructor of a base class.



With next generation of java script we can write the same code without this keyword and constructor at all and with arrow functions:

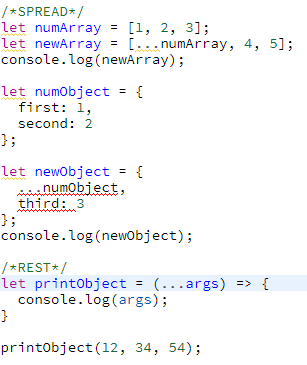


## Spread and Rest

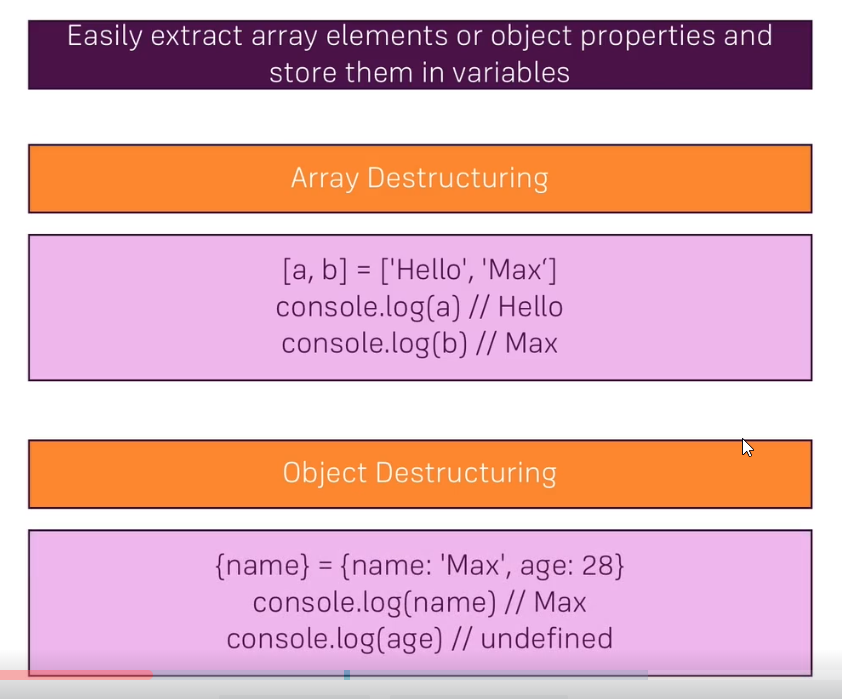
The spread and rest operator is “…”.

As a spread you can extract all the values from some array or object and store it into another array or object with all other values inside this new array or object.

As a rest you can use it in a function as a parameter which receives the list of arguments (like params [] inside .NET)



## Destructuring



When destructuring arrays you need to keep mind on order of elements. With objects you need to pay attention on keys (must match).

## Reference and primitive types

This is the same as in .NET. Reference types are classes, objects, arrays and they are stored into memory with pointers to that location. Value types are creating are not using pointers but they keep the value inside. If you want a copy of reference type, use spread operator.

# Project Set up

## Using React Create-app

We will use tool to set up complete project for us, with all dependences and bundling tools. Instructions on this link <https://github.com/facebook/create-react-app> .

Node.js latest version must be installed prior to any action with the React Create-app.

## First look at created project

It is similar to Angular project. It has only the Index page, and inside that index page you are adding references towards the css and the js files. Index.html has its own Index.js file, which renders App.js component and html code inside. The rendering is executed with this peace of code:

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

App.js file looks like this:

import React, { Component } from 'react';

import './App.css';

class App extends Component {

render() {

return (

<div className="App">

<h1>Hi, I am React App</h1>

</div>

);

}

}

export default App;

It is component which inherits from the Component class. It has a render method that returns content for the index.js to render inside Index.html file. This part of code, inside render method is called jsx.

This is another way of writing component in React:

import React from 'react';

const person = () => {

return (

<p>I am a Person!</p>

);

}

export default person;

## Dynamic content output

In angular for something like this we would use interpolation {{}} but in react we use just one pair of brackets like this

<p>I am a Person and I am {Math.floor(Math.random() \* 30)} years old.</p>

## Dynamic properties

App.js

import React, { Component } from 'react';

import './App.css';

import Person from './Person/Person';

class App extends Component {

render() {

return (

<div className="App">

<h1>Hi, I am React App</h1>

<Person name="Marinko" age="35" />

<Person name="Marina" age="32">My hobby is to love my husband.</Person>

<Person name="Goran" age="28"/>

</div>

);

}

}

export default App;

Person.js

import React from 'react';

const person = (props) => {

return (

<div>

<p>I am a {props.name} and I am {props.age} years old.</p>

<p>{props.children}</p>

</div>

);

}

export default person;

With props.children we are collecting content between open and close Person tags.

## Using state inside component

State is special property that works only inside component that extends Component class. If state changes, complete DOM is rendered again. This is how to use state:

class App extends Component {

state = {

persons: [

{name: "Marinko", age: 35},

{name: "Marina", age: 32},

{name: "Goran", age: 28}

]

}

render() {

return (

<div className="App">

<h1>Hi, I am React App</h1>

<button>Switch button</button>

<Person name={this.state.persons[0].name} age={this.state.persons[0].age} />

<Person name={this.state.persons[1].name} age={this.state.persons[1].age}>My hobby is to love my husband.</Person>

<Person name={this.state.persons[2].name} age={this.state.persons[2].age}/>

</div>

);

}

}

export default App;

## Activating the handler and setting state

switchState = () => {

console.log("clicked");

}

<button onClick={this.switchState}>Switch button</button>

You need to include complete object you want to change with the changed values inside setState function. If you have any other properties inside state object, they won’t be changed at all.

switchState = () => {

this.setState({

persons: [

{ name: "Marinko", age: 35 },

{ name: "Marina", age: 32 },

{ name: "Jovan", age: 29 }

]

});

}

## Transferring handler between components

We can reference handler from one component into another component in the same way as we reference properties. It is sent as an attribute like name and age.

App.js

<Person name={this.state.persons[0].name} age={this.state.persons[0].age} click={this.switchState} />

Person.js

<p onClick={props.click}>I am a {props.name} and I am {props.age} years old.</p>

## Passing params to the handler

switchState = (newName, newAge) => {

this.setState({

persons: [

{ name: "Marinko", age: 35 },

{ name: "Marina", age: 32 },

{ name: newName, age: newAge }

]

});

}

<button onClick={this.switchState.bind(this, "Jovan", 29)}>Switch button</button>

## Two way binding

We can create two way binding by sending an handler to Pearson.js component to call it whenever you want to change something, and also writing it down into text box.

App.js

changeName = (event) => {

this.setState({

persons: [

{ name: "Marinko", age: 35 },

{ name: "Marina", age: 32 },

{ name: event.target.value, age: 29 }

]

});

}

<Person name={this.state.persons[2].name} age={this.state.persons[2].age} change={this.changeName} />

Pearson.js

<input type="text" onChange={props.change} value={props.name} />

## Styling

When using styling it is enough to just create css file, to IMPORT it inside component you want to use it in and to call the class in appropriate element. If using inline style then make an object with css properties and call in in element with attribute style={someStyleObject}.

# List and conditionals

## Conditioning

To do condition rendering you need to wrap all of your code inside curly brackets. That way it can accept java script code. Then you can just use ternary operator.

state = {

persons: [

{ name: "Marinko", age: 35 },

{ name: "Marina", age: 32 },

{ name: "Goran", age: 28 }

],

other: "something other here.",

showPersons: false

}

togglePerson = () => {

this.setState({

showPersons: !this.state.showPersons

})

}

render() {

return (

<div className="App">

<h1>Hi, I am React App</h1>

{/\* <button onClick={this.switchState.bind(this, "Jovan", 29)}>Switch button</button> \*/}

<button onClick={this.togglePerson}>Switch button</button>

{

this.state.showPersons ?

<div>

<Person name={this.state.persons[0].name} age={this.state.persons[0].age} click={this.switchState.bind(this, "Jovan", 29)} />

<Person name={this.state.persons[1].name}

age={this.state.persons[1].age} other={this.state.other}>My hobby is to love my husband.</Person>

<Person name={this.state.persons[2].name} age={this.state.persons[2].age} change={this.changeName} />

</div> : null

}

</div>

);

}

**There is another, better, way of doing this** =>

I can extract all this html inside condition into variable only if condition is true something. Then just invoke this variable inside jsx return between curly brackets like this:

render() {

let persons = null;

if(this.state.showPersons){

persons = (

<div>

<Person name={this.state.persons[0].name} age={this.state.persons[0].age} click={this.switchState.bind(this, "Jovan", 29)} />

<Person name={this.state.persons[1].name}

age={this.state.persons[1].age} other={this.state.other}>My hobby is to love my husband.</Person>

<Person name={this.state.persons[2].name} age={this.state.persons[2].age} change={this.changeName} />

</div>

);

}

return (

<div className="App">

<h1>Hi, I am React App</h1>

{/\* <button onClick={this.switchState.bind(this, "Jovan", 29)}>Switch button</button> \*/}

<button onClick={this.togglePerson}>Toggle persons</button>

{persons}

</div>

);

}

## Lists

To show the list of elements, and not repeating it in jsx part, we can use map method on the persons array inside state object.

render() {

let persons = null;

if (this.state.showPersons) {

persons = (

<div>

{

this.state.persons.map( (person, index) => {

return <Person name={person.name} age={person.age}></Person>

})

}

</div>

);

}

Then we just use this “persons” reference inside jsx code.

When coping array form the reference object you are just making a copy that point to the same location, therefore making mutable change. If you want to make it immutable then you can use spread operator on that array.

this.state.persons.map((person, index) => {

return <Person name={person.name} age={person.age}

click={this.deletePerson.bind(this, index)}></Person>

deletePerson = (personIndex) => {

const persons = [...this.state.persons];

persons.splice(personIndex, 1);

this.setState({ persons: persons })

}

## Keys in the list rendering

It is very important to have a key while rendering html from the list. It helps react a lot to know which element to render. For the key always send unique element inside your object like id.

state = {

persons: [

{ id: 1, name: "Marinko", age: 35 },

{ id: 2, name: "Marina", age: 32 },

{ id: 3, name: "Goran", age: 28 }

],

other: "something other here.",

showPersons: false

}

this.state.persons.map((person, index) => {

return <Person name={person.name} age={person.age}

click={this.deletePerson.bind(this, index)} key={person.id}></Person>

})

## Flexible lists

render() {

let persons = null;

if (this.state.showPersons) {

persons = (

<div>

{

this.state.persons.map((person, index) => {

return <Person name={person.name} age={person.age}

click={this.deletePerson.bind(this, index)} key={person.id}

change={this.changeName.bind(this, person.id)}></Person>

})

}

</div>

);

}

changeName = (id, event) => {

let personIndex = this.state.persons.findIndex(p => p.id === id);

let person = {...this.state.persons[personIndex]};

person.name = event.target.value;

let persons = [... this.state.persons];

persons[personIndex] = person;

this.setState({persons: persons});

}

It is important to know if you send not just event to the function, this event parameter must be on the last place if you send params with bind() method.

# Styling components

## Dynamically styling components

We can define style elements as a json object inside render method, and call it like style attribute inside button or any element.

render() {

const style = {

backgroundColor: 'green',

color: 'white',

font: 'inherit',

border: '1px solid blue',

padding: '8px',

cursor: 'pointer'

};

<button style={style} onClick={this.togglePerson}>Toggle persons</button>

Now we can dynamically change the style in our “if” statement:

if (this.state.showPersons) {

persons = (

<div>

{

this.state.persons.map((person, index) => {

return <Person name={person.name} age={person.age}

click={this.deletePerson.bind(this, index)} key={person.id}

change={this.changeName.bind(this, person.id)}></Person>

})

}

</div>

);

style.backgroundColor = 'red';

}

## Dynamically set classes to style components

ClassName property accepts one string with name of the class or on string with name of the classes blank space separated.

selectClasses = (person) => {

let classArray = [];

if(person.age > 33){

classArray.push('red');

}

else if(person.age < 33 && person.age > 28){

classArray.push('green');

}

else{

classArray.push('bold');

classArray.push('italic');

}

return classArray.join(' ');

}

this.state.persons.map((person, index) => {

let classes = this.selectClasses(person);

return <Person name={person.name} age={person.age}

click={this.deletePerson.bind(this, index)} key={person.id}

change={this.changeName.bind(this, person.id)}

classNames={classes}></Person>

})

Person.js

<p onClick={ props.click } className={props.classNames}>I am { props.name } and I am { props.age } years old.</p>

App.css

.red{ color: red; }

.green{ color: green; }

.bold{ font-weight: bold; }

.italic{ font-style: italic; }

## Using Radium library to create :hover effect on style

Radium is library that allows us to create ‘:hover’ property inside json object, and to be accepted as a style element. We need it if we use inline style object. For me better way is to use classes. Also you can use Radium for media queries.

## CSS Modules

In react, as in contrary to angular, css files could be shared if you use classes with the same name. We need to change configuration of eject: “react-scripts eject”. **BOOTSTRAP won’t work if this is executed.**

Run the command: npm run eject

After operation finishes there will be additional folders in solution. Open webpack.config.dev file and finde css-loader part and change options part:

options: {

importLoaders: 1,

modules: true,

localIdentName: '[name]\_\_[local]\_\_[hash:base64:5]'

},

Then copy the importLoaders, modules and localIdentName to the options part in webpack.config.prod file.

Now every import css needs to have name like **import classes from “./App.css”.** And now all the classes inside the component needs to be called as classes.name\_of\_class

classArray.push(classes.bold);

<div className={classes.App}>

At the end, restart npm in command window, for the changes to take effect.

By the way, if you somehow also want to define a global (i.e. un-transformed) CSS class in such a .css file, you can prefix the selector with :global .

**Example:**

:global .Post { ... }

Now you can use className="Post" anywhere in your app and receive that styling.

# Debugging react apps

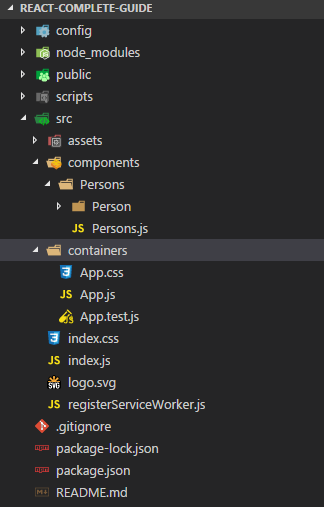
## Debugging part and insights

It is all pretty same as everywhere. Chrome debug window etc. But there is a free tool we can install, react developer tools which could help us a little. Also for production if you think the component could fall apart you could wrap it inside <ErrorBoundary></ErrorBoundary> peace of code. For how to create that look at Section 6 part 75 of the tutorial.

# Components deep dive

## Structuring components in better order

It is always much better to have structure that is easier to maintain and to search through. Creating components, containers, assets folders will help to that.



## Splitting into smaller components

In persons component we will extract generation of Pearson list. Also extract selectClasses function and some css classes like this:

import React from 'react';

import Person from './Person/Person';

import classes from './Persons.css';

const selectClasses = (person) => {

let classArray = [];

if (person.age > 33) {

classArray.push(classes.red);

}

else if (person.age < 33 && person.age > 28) {

classArray.push(classes.green);

}

else {

classArray.push(classes.bold);

classArray.push(classes.italic);

}

return classArray.join(' ');

}

const persons = (props) => {

return (

props.persons.map((person, index) => {

return <Person name={person.name} age={person.age}

click={props.deletePerson.bind(this, index)} key={person.id}

change={props.changeName.bind(this, person.id)}

classNames={selectClasses(person)} />

})

);

}

export default persons;

App.js

let persons = null;

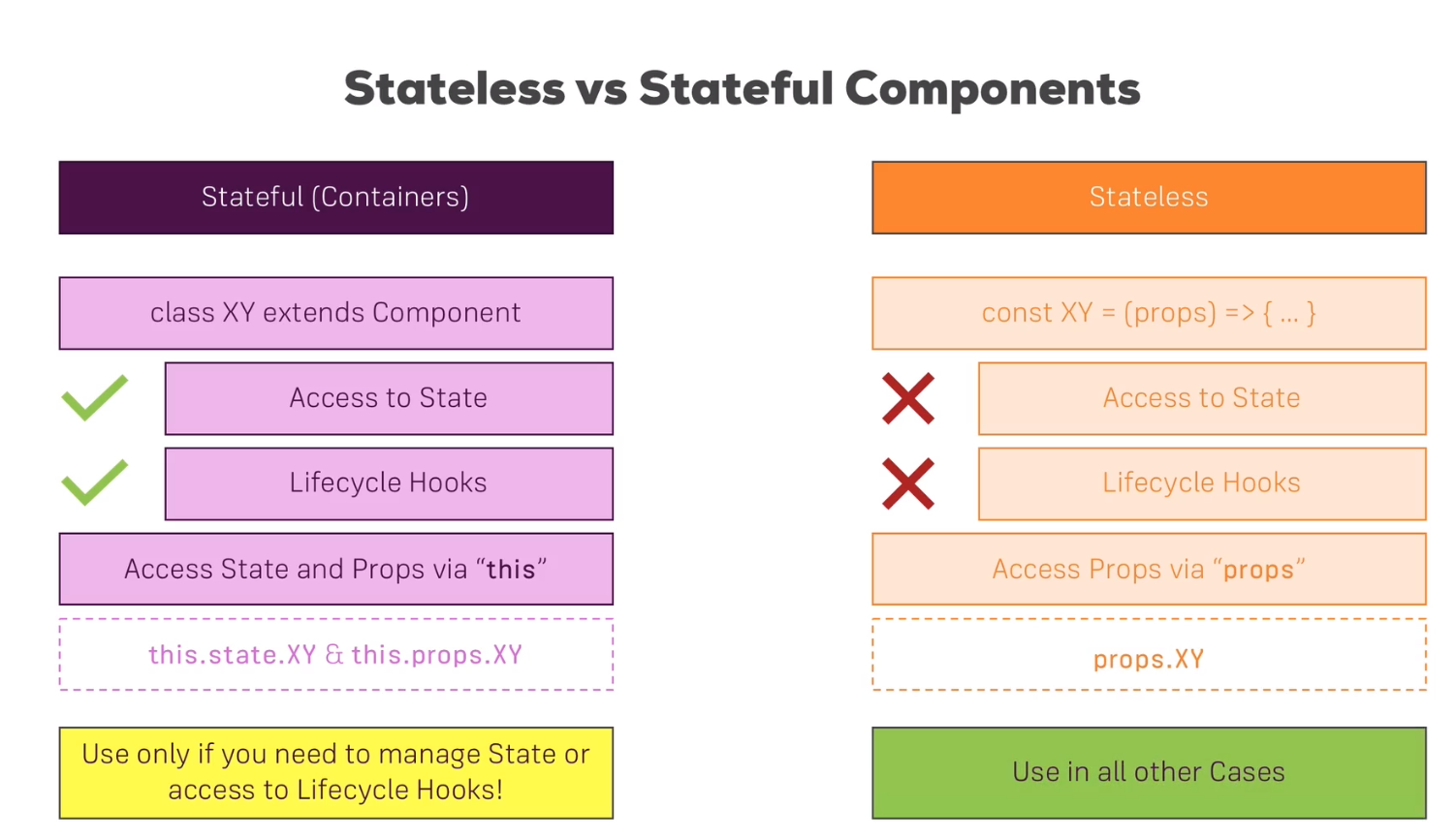
if (this.state.showPersons) {

persons = <Persons persons={this.state.persons} deletePerson={this.deletePerson}

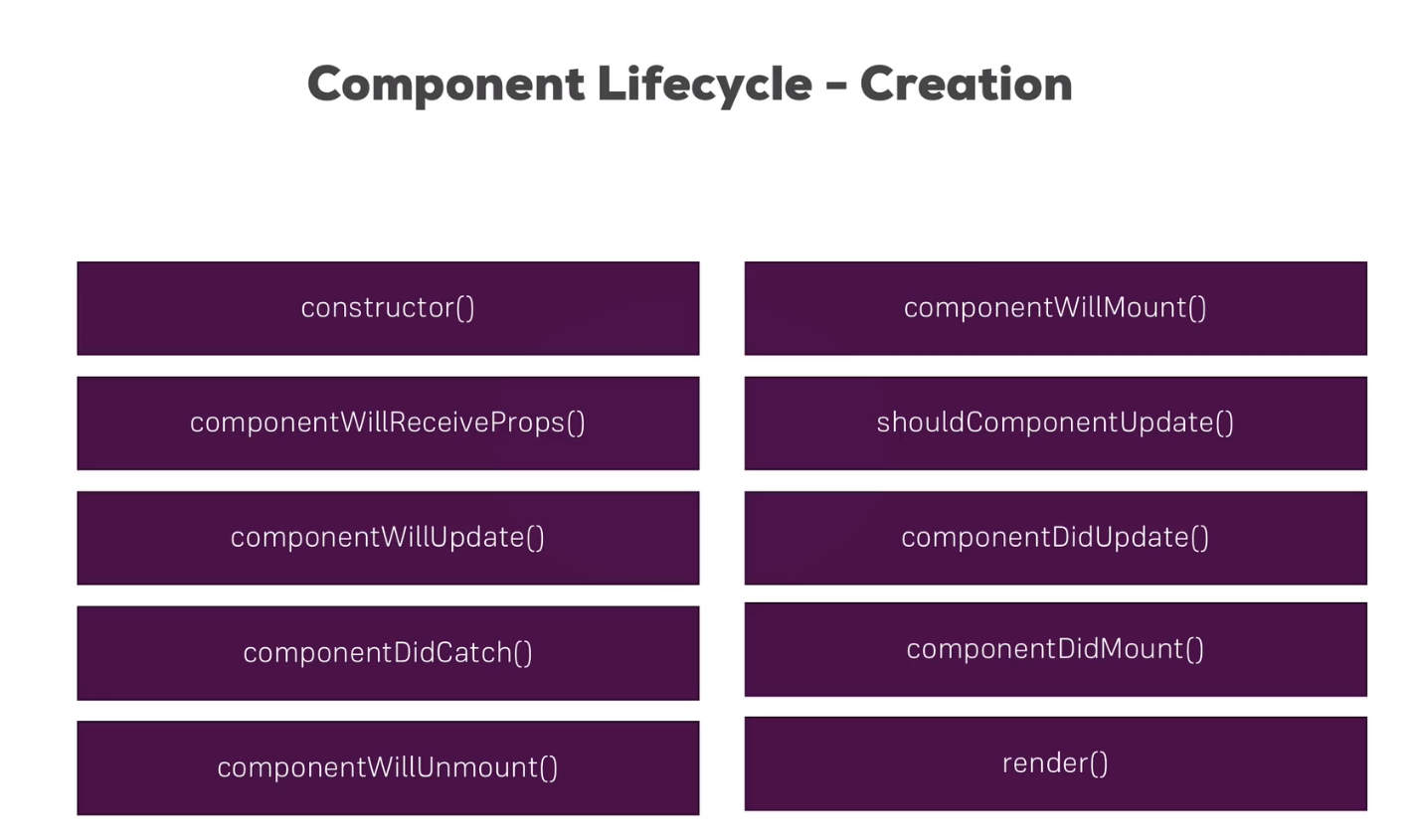
changeName={this.changeName} />

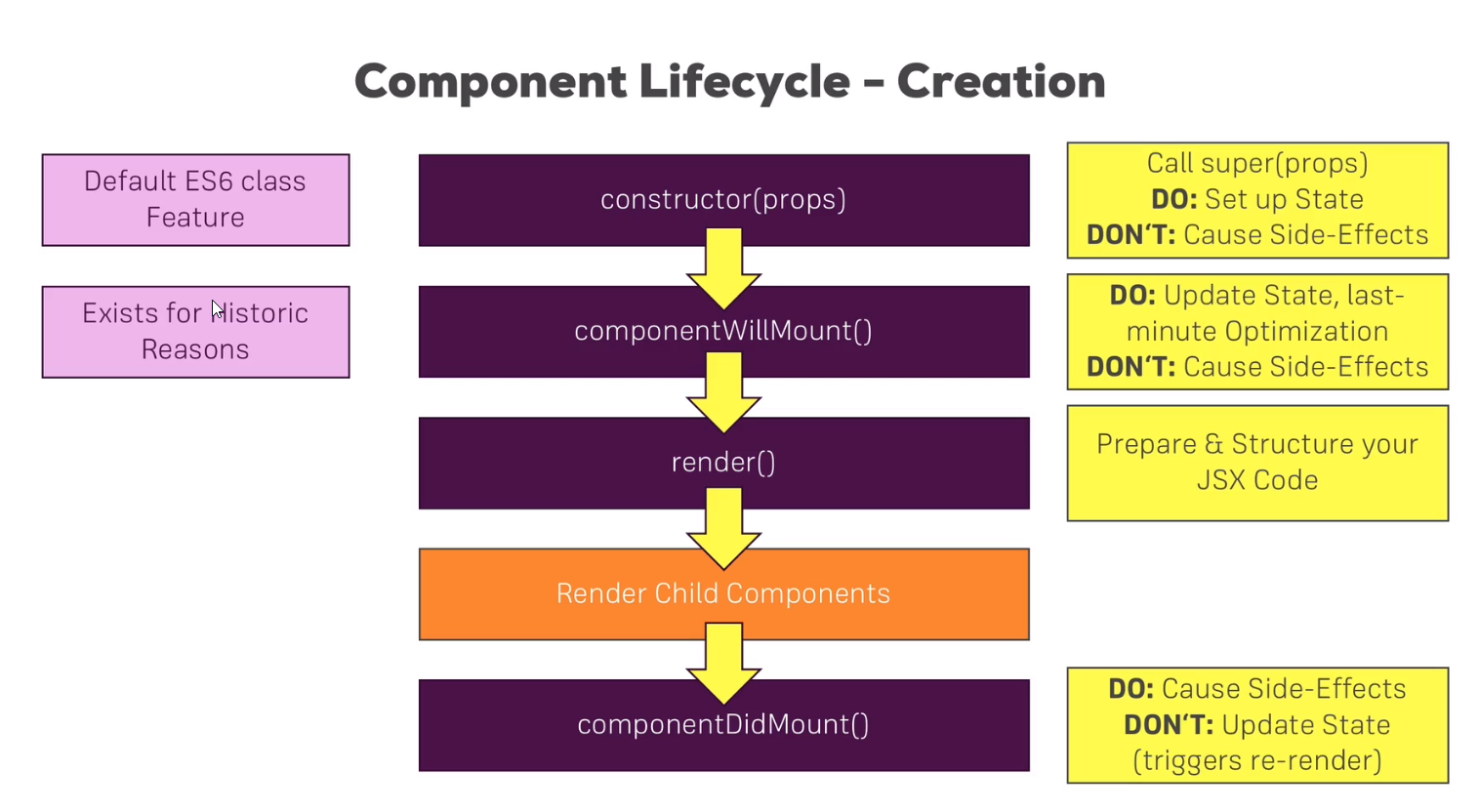
The important thing to know is that functions that changes the state needs to stay inside container, in this case in App.js. All the functions that doesn’t change the state, and are related to the imported component, should be removed from container and placed inside imported component as we did width selectClasses.

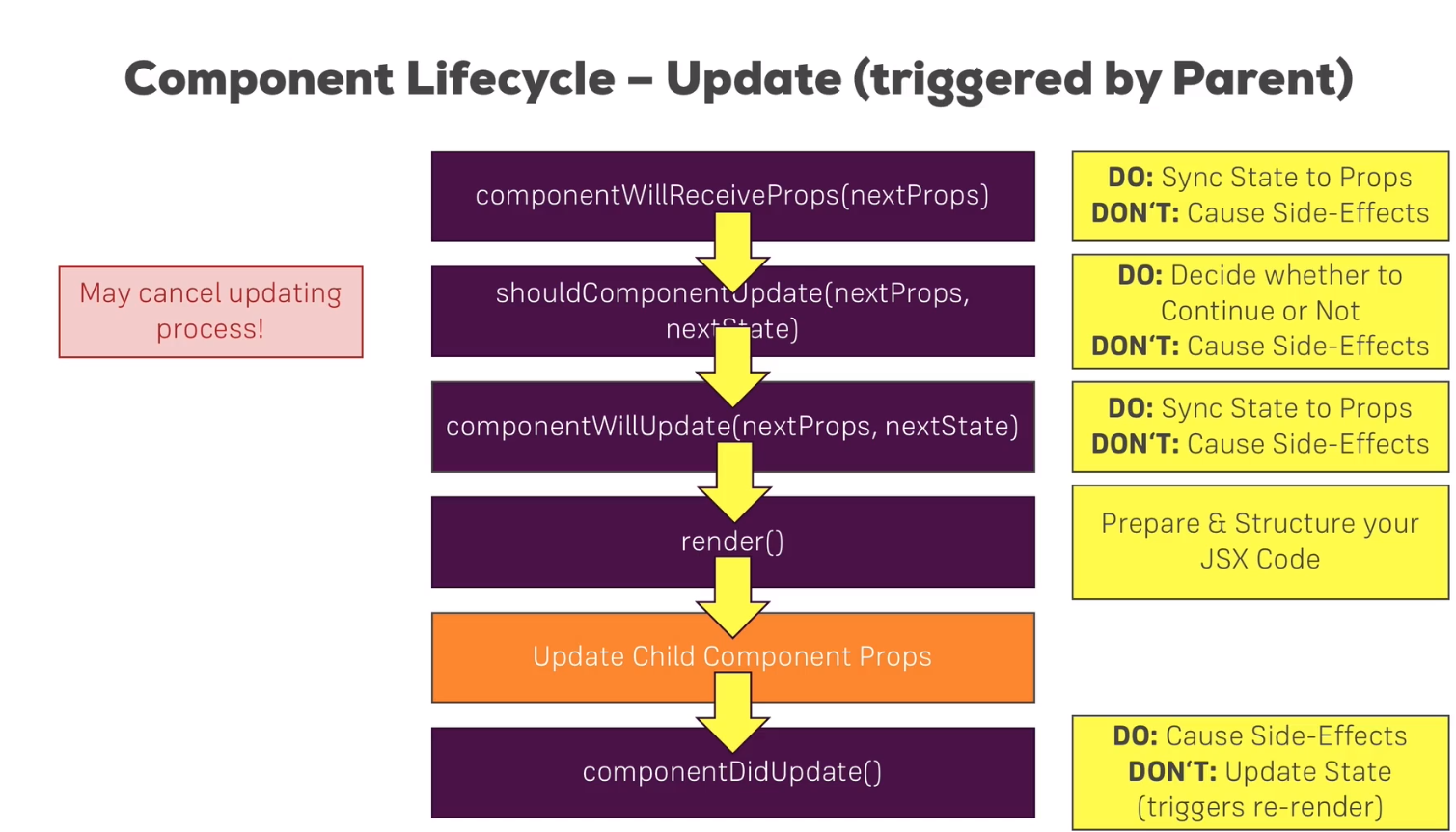
## Stateful vs Stateless components



## Lifecycle in react







## Aux implementation (Higher order component – hoc)

In the cases as in our Cockpit component, when you have only elements wrapped inside div tag, just because jsx doesn’t accept elements without root element, you can remove that div wrapper and replace it with Aux implementation. **Don’t name the file Aux.js it could lead to problem.**

Old Cockpit component:

const cockpit = (props) => {

return (

<div>

<h1>{props.title}</h1>

{/\* <button onClick={this.switchState.bind(this, "Jovan", 29)}>Switch button</button> \*/}

<button className={props.buttonClass} onClick={props.togglePerson}>Toggle persons</button>

</div>

);

}

Auxiliary.js

const auxiliary = (props) => props.children;

export default auxiliary;

New Cockpit component

import Aux from '../../hoc/Auxiliary';

const cockpit = (props) => {

return (

<Aux>

<h1>{props.title}</h1>

{/\* <button onClick={this.switchState.bind(this, "Jovan", 29)}>Switch button</button> \*/}

<button className={props.buttonClass} onClick={props.togglePerson}>Toggle persons</button>

</Aux>

);

}

This way in the DOM you won’t find another div element that can screw up your css in the app.

## SetState method properly

SetState method is always executed asynchrony therefore never use states properties inside setstate method. If you need to have some calculations you can always extract states properties with spread if they are referenced types or without if they are value types and then update them and after that update state, as we did in above examples with delete person or change name of person.

state = {

persons: [

{ id: 1, name: "Marinko", age: 35 },

{ id: 2, name: "Marina", age: 32 },

{ id: 3, name: "Goran", age: 28 }

],

other: "something other here.",

showPersons: false,

countToggle: 0

}

togglePerson = () => {

let count = this.state.countToggle;

this.setState({

showPersons: !this.state.showPersons,

countToggle: count + 1

})

}

Or you can write this kind of method if you need something like this we have done right above

togglePerson = () => {

this.setState((prevState, props) => {

return {

showPersons: !this.state.showPersons,

countToggle: prevState.countToggle + 1

}

})

}

In the prevState parameter we are saving the new state which is not pointing to old one, thus it I safe to use it in asynch method.

## Wrapping component with hoc

We can even wrap our entire component with hoc component something like this.

WithClass.js => pay attention on {…props} part. Without that we wouldn’t see our title.

import React from 'react';

const withClass = (WrappedComponent, className) => {

return(props) => (

<div className={className}>

<WrappedComponent {...props} />

</div>

)

}

export default withClass;

Then we need to import it and aux into the component we want to wrap:

import Aux from '../hoc/Auxiliary';

import WithClass from '../hoc/WithClass';

return (

<Aux>

<Cockpit buttonClass={buttonClass} togglePerson={this.togglePerson} title={this.props.title} />

{persons}

</Aux>

);

export default WithClass(App, classes.App);

# Project

## Basic setup

We have created aux component, then Layout to handle navigation menu and body for every menu, and all that imported into app.js file.

Aux.js

const aux = (props) => props.children;

export default aux;

Layout.js

import React from 'react';

import Aux from '../../hoc/Auxiliaru';

const layout = (props) => {

return (

<Aux>

<div>Toolbar, SideBar, Backdrop</div>

<main>

{props.children}

</main>

</Aux>

)

};

export default layout;

App.js

import React, { Component } from 'react';

import Layout from '../components/Layout/Layout';

class App extends Component {

render() {

return (

<div>

<Layout>

<p>Test</p>

</Layout>

</div>

);

}

}

export default App;

## BurgerBuilder container

We are adding a new container folder and inside it BurgerBuilder folder. Then we create this Stateful component

import React, { Component } from 'react';

import Aux from '../../hoc/Auxiliaru';

class BurgerBuilder extends Component {

render() {

return(

<Aux>

<div>Burger</div>

<div>Build Controls</div>

</Aux>

);

}

}

export default BurgerBuilder;

And then change app.js

<div>

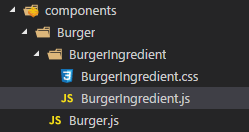
<Layout>

<BurgerBuilder />

</Layout>

</div>

Then we are continuing with the following structure:



The BurgerIngredient.js looks like this:

import React from 'react';

import classes from './BurgerIngredient.css';

const burgerIngredient = (props) => {

let ingredient = null;

switch (props.type) {

case ('bread-bottom'):

ingredient = <div className={classes.BreadBottom}></div>;

break;

case ('bread-top'):

ingredient = (

<div className={classes.BreadTop}>

<div className={classes.Seeds1}></div>

<div className={classes.Seeds2}></div>

</div>

);

break;

case ('meat'):

ingredient = <div className={classes.Meat}></div>;

break;

case ('cheese'):

ingredient = <div className={classes.Cheese}></div>;

break;

case ('bacon'):

ingredient = <div className={classes.Bacon}></div>;

break;

case ('salad'):

ingredient = <div className={classes.Salad}></div>;

break;

default: null;

return ingredient;

}

}

export default burgerIngredient;

We also have a BurgerIngredient.css file which contains all those classes.

## Prop type validation

To install: npm install –save prop-types.

For prop-types to work with our function we need to convert it to class.

import React, { Component } from 'react';

import PropTypes from 'prop-types';

import classes from './BurgerIngredient.css';

class BurgerIngredient extends Component {

render() {

let ingredient = null;

switch (props.type) {

case ('bread-bottom'):

ingredient = <div className={classes.BreadBottom}></div>;

break;

case ('bread-top'):

ingredient = (

<div className={classes.BreadTop}>

<div className={classes.Seeds1}></div>

<div className={classes.Seeds2}></div>

</div>

);

break;

case ('meat'):

ingredient = <div className={classes.Meat}></div>;

break;

case ('cheese'):

ingredient = <div className={classes.Cheese}></div>;

break;

case ('bacon'):

ingredient = <div className={classes.Bacon}></div>;

break;

case ('salad'):

ingredient = <div className={classes.Salad}></div>;

break;

default: null;

return ingredient;

}

}

}

BurgerIngredient.propTypes = {

type: PropTypess.string.isRequired

};

export default BurgerIngredient;

## Burger component

Burger.js file

import React from 'react';

import classes from './Burger.css';

import BurgerIngredient from './BurgerIngredient/BurgerIngredient';

const burger = (props) => {

return (

<div className={classes.Burger}>

<BurgerIngredient type="bread-top" />

<BurgerIngredient type="cheese" />

<BurgerIngredient type="meat" />

<BurgerIngredient type="bread-bottom" />

</div>

);

}

export default burger;

Burger.css

.Burger{

width: 100%;

margin: auto;

height: 250px;

overflow: auto;

text-align:center;

font-weight: bold;

font-size: 1.2rem;

}

@media(min-width: 1000px) and (min-height: 700px){

.Burger{

width: 700px;

height: 600px;

}

}

@media(min-width: 500px) and (min-height: 401px){

.Burger{

width: 450px;

height: 400px;

}

}

@media(min-width: 500px) and (min-height: 400px){

.Burger{

width: 350px;

height: 300px;

}

}

## Introducing state with array transformation

The state will use as for ingredients

class BurgerBuilder extends Component {

state = {

ingredients: {

salad: 1,

bacon: 1,

cheese: 2,

meat: 2

}

}

render() {

return(

<Aux>

<Burger ingredients={this.state.ingredients} />

<div>Build Controls</div>

</Aux>

);

}

}

export default BurgerBuilder;

burger.js file

const burger = (props) => {

let transformedIngredients = Object.keys(props.ingredients) //this Object.keys just extract the keys from the object to the array

.map((ingKey, index) => {

return [...Array(props.ingredients[ingKey])]

.map((\_, ind) => {

return <BurgerIngredient type={ingKey} key={ingKey + ind} />

});

});

return (

<div className={classes.Burger}>

<BurgerIngredient type="bread-top" />

{transformedIngredients}

<BurgerIngredient type="bread-bottom" />

</div>

);

}

export default burger;

## Using reduce method on array

We have changed the burger.js component like this to check if arrays has any elements at all

const burger = (props) => {

let transformedIngredients = Object.keys(props.ingredients) //this Object.keys just extract the keys from the object to the array

.map((ingKey, index) => {

return [...Array(props.ingredients[ingKey])]

.map((\_, ind) => {

return <BurgerIngredient type={ingKey} key={ingKey + ind} />

});

}).reduce((arr, el) => { //fancy way of placing values from all the arrays to one array

return arr.concat(el);

}, []);

if(!transformedIngredients.length){

transformedIngredients = <p>Please start adding ingredients</p>

}

return (

<div className={classes.Burger}>

<BurgerIngredient type="bread-top" />

{transformedIngredients}

<BurgerIngredient type="bread-bottom" />

</div>

);

}

We have changed all the values in state.ingredients object to 0.

## Creating build controls

import React from 'react';

import classes from './BuildControl.css';

const buildControl = (props) => {

return (

<div className={classes.BuildControl}>

<div className={classes.Label}>{props.label}</div>

<button className={classes.Less}>Less</button>

<button className={classes.More}>More</button>

</div>

);

}

export default buildControl;

import React from 'react';

import classes from './BuildControls.css';

import BuildControl from '../BuildControls/BuildControl/BuildControl';

const controls = [

{ label: 'Salad', type: 'salad' },

{ label: 'Bacon', type: 'bacon' },

{ label: 'Cheese', type: 'cheese' },

{ label: 'Meat', type: 'meat' }

];

const buildControls = (props) => {

return (

<div className={classes.BuildControls}>

{

controls.map( control => {

return <BuildControl key={control.label} label={control.label} />

})

}

</div>

);

}

export default buildControls;

import React, { Component } from 'react';

import Aux from '../../hoc/Auxiliaru';

import Burger from '../../components/Burger/Burger';

import BuildControls from '../../components/Burger/BuildControls/BuildControls';

class BurgerBuilder extends Component {

state = {

ingredients: {

salad: 0,

bacon: 0,

cheese: 0,

meat: 0

}

}

render() {

return(

<Aux>

<Burger ingredients={this.state.ingredients} />

<BuildControls />

</Aux>

);

}

}

export default BurgerBuilder;

## Add and remove ingredients

In BurgerBuilder.js we create one object and three methods to manipulate adding and removing of the ingredients:

ingredientPrice = {

salad: 0.4,

bacon: 0.8,

cheese: 0.6,

meat: 1.3

}

addIngredient = (type) => {

let newIngredients = { ...this.state.ingredients };

let newTotal = this.state.totalPrice;

newIngredients[type] = newIngredients[type] + 1;

newTotal = newTotal + this.ingredientPrice[type];

this.setState({ ingredients: newIngredients, totalPrice: newTotal });

}

removeIngredient = (type) => {

if (this.state.ingredients[type] > 0) {

let newIngredients = { ...this.state.ingredients };

let newTotal = this.state.totalPrice;

newIngredients[type] = newIngredients[type] - 1;

newTotal = newTotal - this.ingredientPrice[type];

this.setState({ ingredients: newIngredients, totalPrice: newTotal });

}

}

shoudDisableButton = () => {

let ingredients = {...this.state.ingredients};

for(let key in ingredients){

ingredients[key] = ingredients[key] <= 0;

}

return ingredients;

}

render() {

return (

<Aux>

<Burger ingredients={this.state.ingredients} />

<BuildControls addIngredient={this.addIngredient} removeIngredient={this.removeIngredient}

disable={this.shoudDisableButton()} />

</Aux>

);

}

Then in all other components we just subscribe to attributes inside BuildControls tag(component)

const controls = [

{ label: 'Salad', type: 'salad' },

{ label: 'Bacon', type: 'bacon' },

{ label: 'Cheese', type: 'cheese' },

{ label: 'Meat', type: 'meat' }

];

const buildControls = (props) => {

return (

<div className={classes.BuildControls}>

{

controls.map( control => {

return <BuildControl key={control.label} label={control.label}

add={props.addIngredient.bind(this, control.type)}

remove={props.removeIngredient.bind(this, control.type)}

disable={props.disable[control.type]}/>

})

}

</div>

);

}

const buildControl = (props) => {

return (

<div className={classes.BuildControl}>

<div className={classes.Label}>{props.label}</div>

<button className={classes.Less} onClick={props.remove} disabled={props.disable}>Less</button>

<button className={classes.More} onClick={props.add}>More</button>

</div>

);

}

## Showing the total price

In BurgerBuilder.js we add just additional property price

render() {

return (

<Aux>

<Burger ingredients={this.state.ingredients} />

<BuildControls addIngredient={this.addIngredient} removeIngredient={this.removeIngredient}

disable={this.shoudDisableButton()} price={this.state.totalPrice} />

</Aux>

);

And in BuildControls.js we subscribe to that attribute:

const buildControls = (props) => {

return (

<div className={classes.BuildControls}>

<p>Current price: <strong>{props.price.toFixed(2)}</strong></p>

{

controls.map( control => {

return <BuildControl key={control.label} label={control.label}

add={props.addIngredient.bind(this, control.type)}

remove={props.removeIngredient.bind(this, control.type)}

disable={props.disable[control.type]}/>

})

}

</div>

);

}

## Purchasable button and logic for disabling it

BurgerBuilder.js

updatePurchaseState = (ingredients) => {

let sum = 0;

Object.keys(ingredients).map((ingKey) => {

return sum = sum + ingredients[ingKey];

})

this.setState({purchasable: sum > 0})

}

This method decides should the purchase button be enabled or disabled. We are calling this method after each call of add or remove ingredient method. Also we have added a new property to state, purchasable.

<Aux>

<Burger ingredients={this.state.ingredients} />

<BuildControls addIngredient={this.addIngredient} removeIngredient={this.removeIngredient}

disable={this.shoudDisableButton()} price={this.state.totalPrice}

purchasable={this.state.purchasable} />

</Aux>

BuilderControls.js

const buildControls = (props) => {

return (

<div className={classes.BuildControls}>

<p>Current price: <strong>{props.price.toFixed(2)}</strong></p>

{

controls.map( control => {

return <BuildControl key={control.label} label={control.label}

add={props.addIngredient.bind(this, control.type)}

remove={props.removeIngredient.bind(this, control.type)}

disable={props.disable[control.type]}/>

})

}

<button disabled={!props.purchasable} className={classes.OrderButton}>ORDER NOW</button>

</div>

);

## Adding modal window

Modal.js file

import React from 'react';

import classes from './Modal.css';

const modal = (props) => {

return (

<div className={classes.Modal}>

{props.children}

</div>

);

}

export default modal;

OrderSummary.js

import React from 'react';

import classes from './OrderSummary.css';

import Aux from '../../../hoc/Auxiliaru';

const orderSummary = (props) => {

let ingSummary = Object.keys(props.ingredients)

.map(igKey => {

return <li key={igKey}>

<span className={classes.AllCaps}>{igKey}</span>: {props.ingredients[igKey]}

</li>

})

return (

<Aux>

<h3>Your order:</h3>

<p>A delicious burger with following ingredients:</p>

<ul>

{ingSummary}

</ul>

<p>Continue to checkout?</p>

</Aux>

);

}

export default orderSummary;

BurgerBuilder.js

render() {

return (

<Aux>

<Modal><OrderSummary ingredients={this.state.ingredients} /></Modal>

<Burger ingredients={this.state.ingredients} />

<BuildControls addIngredient={this.addIngredient}

removeIngredient={this.removeIngredient}

disable={this.shoudDisableButton()} price={this.state.totalPrice}

purchasable={this.state.purchasable} />

</Aux>

);

}

## Showing Modal window

class BurgerBuilder extends Component {

state = {

ingredients: {

salad: 0,

bacon: 0,

cheese: 0,

meat: 0

},

totalPrice: 4,

purchasable: false,

purchasing: false

}

…

purchaseHandler = () => {

this.setState({ purchasing: true });

}

render() {

return (

<Aux>

<Modal show={this.state.purchasing}>

<OrderSummary ingredients={this.state.ingredients} />

</Modal>

<Burger ingredients={this.state.ingredients} />

<BuildControls

addIngredient={this.addIngredient}

removeIngredient={this.removeIngredient}

disable={this.shoudDisableButton()} price={this.state.totalPrice}

purchasable={this.state.purchasable} ordered={this.purchaseHandler} />

</Aux>

);

}

BuildControls.js

<button disabled={!props.purchasable} className={classes.OrderButton}

onClick={props.ordered}>ORDER NOW</button>

Modal.js

import React from 'react';

import classes from './Modal.css';

const renderClasses = (show) => {

let classesArray = [];

if(show){

classesArray.push(classes.Modal, classes.ShowModal);

}

else{

classesArray.push(classes.Modal, classes.HideModal);

}

return classesArray.join(' ');

}

const modal = (props) => {

let modalClasses = renderClasses(props.show);

return (

<div className={modalClasses}>

{props.children}

</div>

);

}

export default modal;

## Backdrop component

With backdrop component we will create the cover screen all over the screen and behind the modal. And also we will enable click on that backdrop to close the modal.

Backdrop.js

import React from 'react';

import classes from './Backdrop.css';

const backdrop = (props) => {

return (

props.show ? <div className={classes.Backdrop} onClick={props.closeOrder}></div> : null

);

}

export default backdrop;

Modal.js

const modal = (props) => {

let modalClasses = renderClasses(props.show);

return (

<Aux>

<BackDrop show={props.show} closeOrder={props.closeOrder}/>

<div className={modalClasses}>

{props.children}

</div>

</Aux>

);

}

BurgerBuilder.js

purchaseCancel = () => {

this.setState({ purchasing: false });

}

render() {

return (

<Aux>

<Modal show={this.state.purchasing} closeOrder={this.purchaseCancel} >

<OrderSummary ingredients={this.state.ingredients} />

</Modal>

<Burger ingredients={this.state.ingredients} />

<BuildControls

addIngredient={this.addIngredient}

removeIngredient={this.removeIngredient}

disable={this.shoudDisableButton()} price={this.state.totalPrice}

purchasable={this.state.purchasable} ordered={this.purchaseHandler} />

</Aux>

);

}

## Creating buttons for the modal (Order summary)

Created new Button component to wrap around regular button

import React from 'react';

import classes from './Button.css';

const button = (props) => {

return (

<button className={[classes.Button, classes[props.buttonType]].join(' ')}

onClick={props.clicked}>{props.children}</button>

);

}

export default button;

In Aux tag of order summary we add two buttons and total price

<p><strong>Total price: {props.totalPrice.toFixed(2)}</strong></p>

<p>Continue to checkout?</p>

<Button buttonType={'Danger'} clicked={props.closeOrder}>CANCEL</Button>

<Button buttonType={'Success'} clicked={props.continueOrder}>SUCCESS</Button>

In BurgerBuilder.js we just add another function and pass those handlers

purchaseContinue = () => {

alert('You have continued!!!');

}

render() {

return (

<Aux>

<Modal show={this.state.purchasing} closeOrder={this.purchaseCancel} >

<OrderSummary ingredients={this.state.ingredients}

closeOrder={this.purchaseCancel} continueOrder={this.purchaseContinue}

totalPrice={this.state.totalPrice}/>

</Modal>

## Toolbar

In the components folder we create Navigation and inside that Toolbar folder

import React from 'react';

import classes from './Toolbar.css';

const toolbar = (props) => {

return(

<header className={classes.Toolbar}>

<div>MENU</div>

<div>LOGO</div>

<nav>

...

</nav>

</header>

);

}

export default toolbar;

In the Layout file we make this change

const layout = (props) => {

return (

<Aux>

<Toolbar />

<main className={classes.content}>

{props.children}

</main>

</Aux>

)

};

Layout.css

.content{

margin-top: 72px;

}

## Adding the LOGO

In the components folder we create the Logo folder

import React from 'react';

import classes from './Logo.css';

import burgerLogo from '../../assets/images/burger-logo.png';

const logo = (props) => {

return(

<div className={classes.Logo}>

<img src={burgerLogo} alt='burger logo'/>

</div>

);

}

export default logo;

This is the way to use img src tag. We need to import it dynamically.

Toolbar.js

const toolbar = (props) => {

return(

<header className={classes.Toolbar}>

<div>MENU</div>

<Logo/>

<nav>