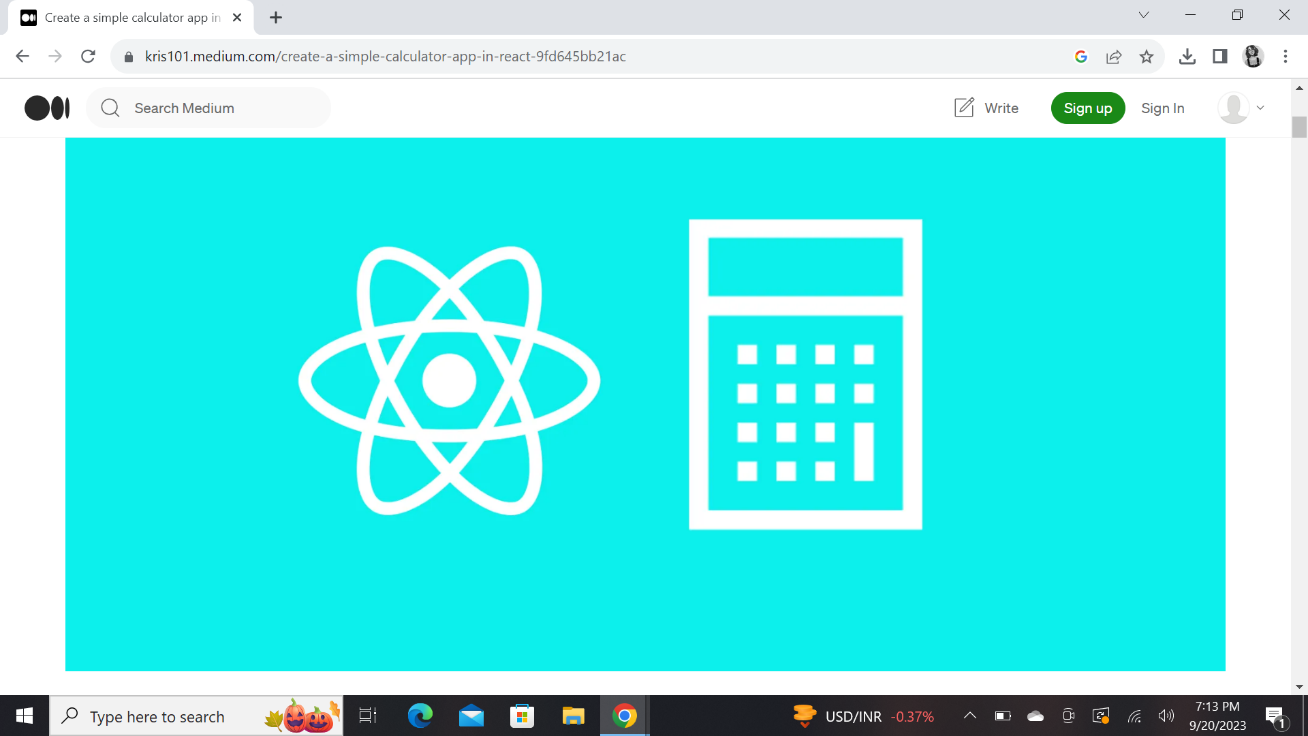
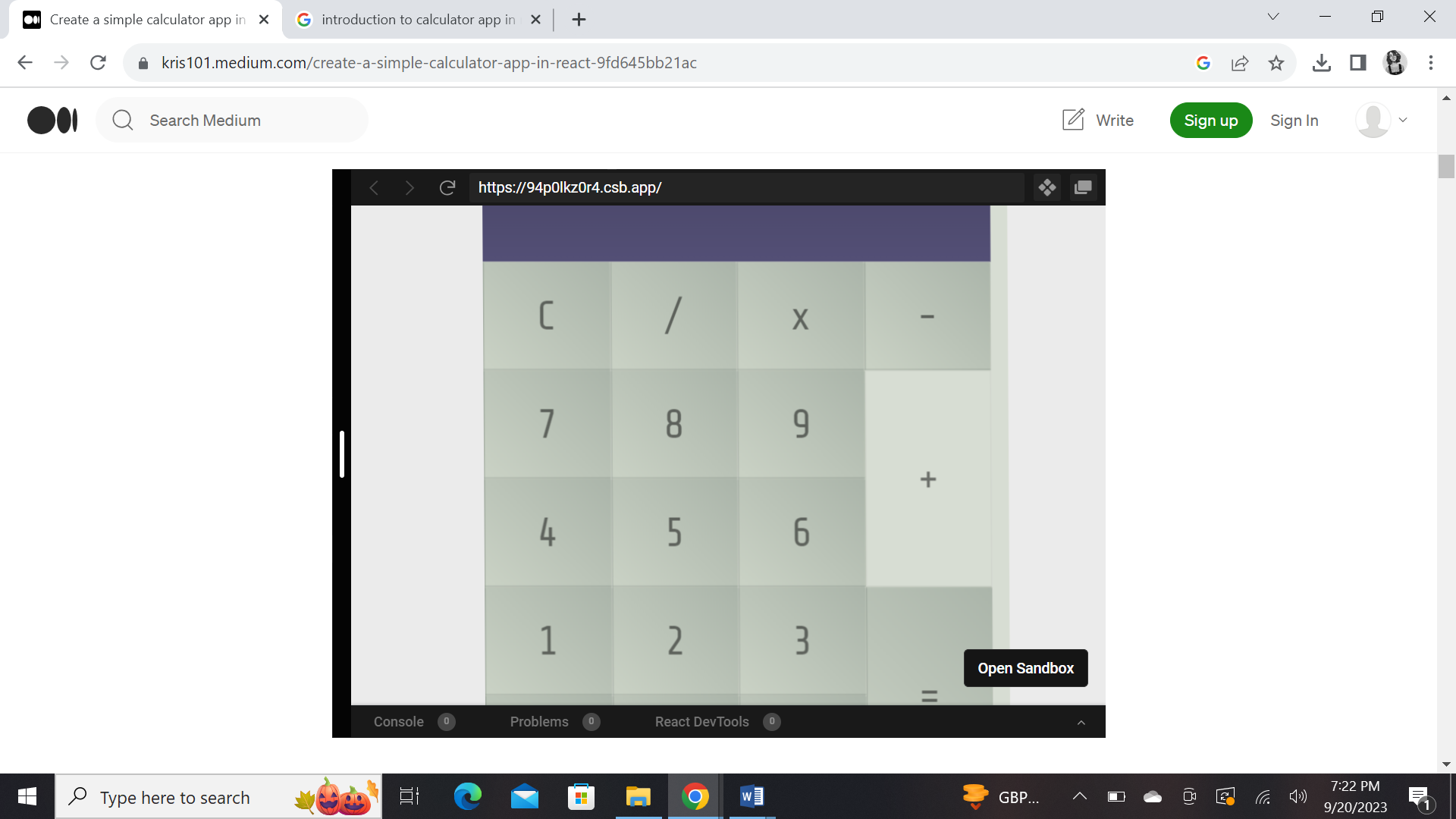
We’ll be creating a new react app, a calculator. While the operations are very simple, it’s a bit of struggle to fix things in place and maintain a proper state.

We got to know what React web apps are, React Components, Props, States, and the lifecycle of a React component. We also created a basic clock application using all of the following. But React is a javascript library to build flexible User Interfaces, and we have not created any so far. So it is time for us to step into the real world of React by creating a simple yet impactful project.

Our aim for this article will be to introduce our readers to a newer idea of learning on the go while developing with us, and we have chosen to create a React Calculator as our first-ever project. Nothing could have been a better way to count the miles to travel with a great start like this.



Here’s the sample of the app we’ll be creating.



create-react-app is where we begin to make sure you have node, npm, create-react-app as usual.

If you don’t, install NodeJS (comes with npm) from here.

Install create-react-app with npm install create-react-app --global on a terminal.

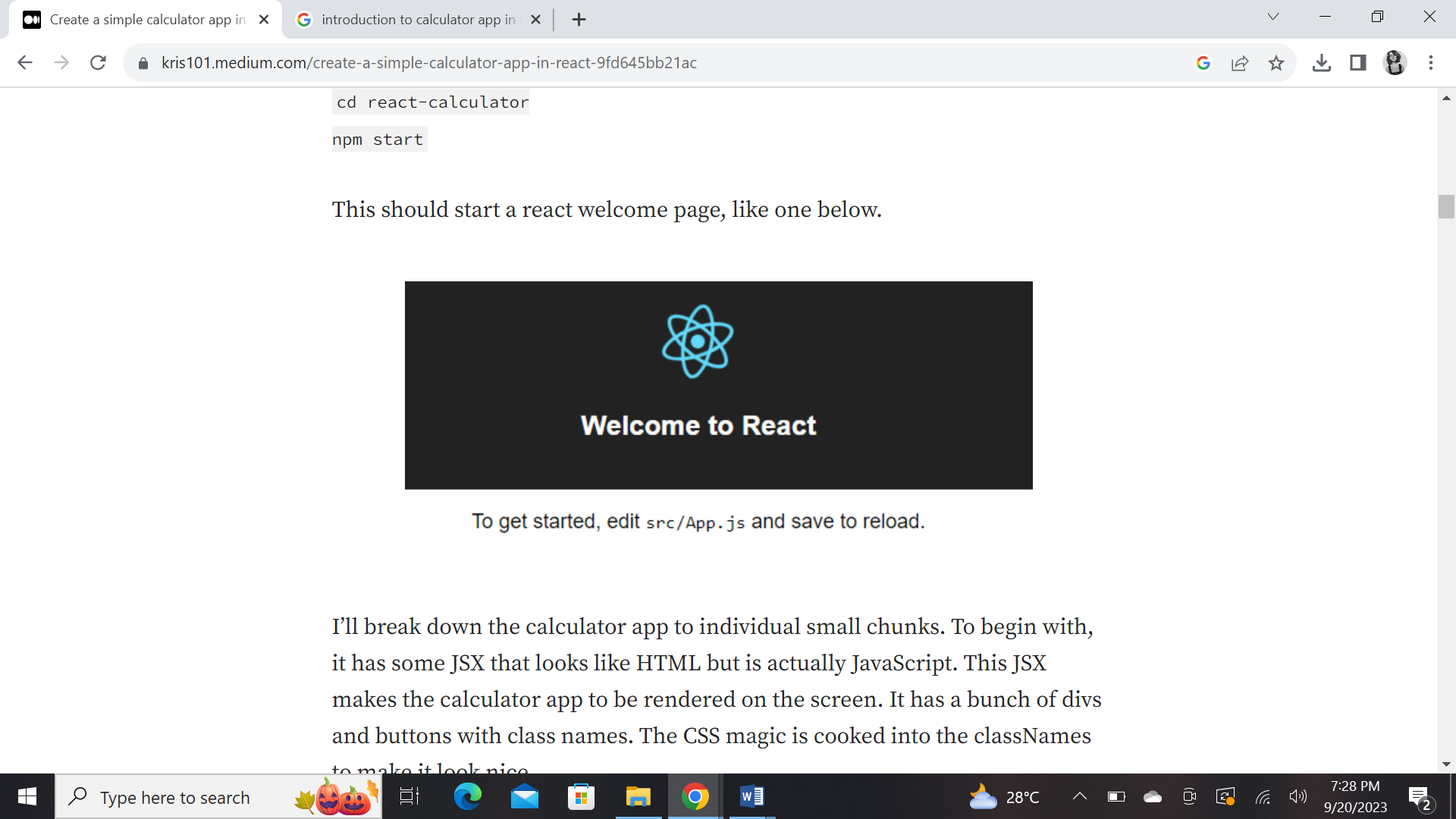
That is all the installation part.

Create a new react app with create-react-app react-calculator. Make sure not to have space in your app name, beginners make this mistake frequently.

cd react-calculator

npm start

This should start a react welcome page, like one below.



I’ll break down the calculator app to individual small chunks. To begin with, it has some JSX that looks like HTML but is actually JavaScript. This JSX makes the calculator app to be rendered on the screen. It has a bunch of divs and buttons with class names. The CSS magic is cooked into the classNames to make it look nice.

**Libraries used to create Calculator in React**

* react, react-dom
* mathjs
* immutability-helper

How to begin?

I started with create-react-app and continued adding components. Now, the app has following components:

* App — main placeholder component
* Buttons — placeholder for an individual button
* Button — individual button
* Display — the display bar on top

**Handling calculator components**

Our calculator performs addition, subtraction, multiplication and division operations. We’ll store all the user input in a variable and calculate on entering equal to (=) sign.

Since input variable has to be modified at every press of a button, we’ll use the state to handle this.

*What is the state in React?*

*The state in react refers to the data state in a component. If the same state is used to render multiple parts in the page, all values will be updated when it changes from anywhere in the app. We use this to store the button presses and diplay values in the display bar*.

We must initialize the state, we do this via a constructor.

class App extends Component {

constructor() {

super()

this.state = { operations: [] }

}

.....

}

The operations variable is the array of button inputs, it may look like [2, \*,4, -, 1].

We have the numeric buttons 0–9 and operators +, -, \*, /. We also have clear, equals and decimal buttons. Each button has a click handler.

render() {

return (

<div className="App">

<Display data={this.state.operations} />

<Buttons>

<Button onClick={this.handleClick} label="C" value="clear" />

<Button onClick={this.handleClick} label="7" value="7" />

<Button onClick={this.handleClick} label="4" value="4" />

<Button onClick={this.handleClick} label="1" value="1" />

<Button onClick={this.handleClick} label="0" value="0" />

<Button onClick={this.handleClick} label="/" value="/" />

<Button onClick={this.handleClick} label="8" value="8" />

<Button onClick={this.handleClick} label="5" value="5" />

<Button onClick={this.handleClick} label="2" value="2" />

<Button onClick={this.handleClick} label="." value="." />

<Button onClick={this.handleClick} label="x" value="\*" />

<Button onClick={this.handleClick} label="9" value="9" />

<Button onClick={this.handleClick} label="6" value="6" />

<Button onClick={this.handleClick} label="3" value="3" />

<Button label="" value="null" />

<Button onClick={this.handleClick} label="-" value="-" />

<Button onClick={this.handleClick} label="+" size="2" value="+" />

<Button onClick={this.handleClick} label="=" size="2" value="equal" />

</Buttons>

</div>

)

It returns a Buttons which holds all the buttons together.

class Buttons extends Component {

render() {

return <div className="Buttons"> {this.props.children} </div>

}

}

The props.children hold everything that’s passed between the <Buttons> </Buttons>, which are all buttons. I have a post planned that’s coming soon on details about props.children. So it just renders the passed button as it is.

The button is, well, the individual button.

class Button extends Component {

render() {

return (

<div

onClick={this.props.onClick}

className="Button"

data-size={this.props.size}

data-value={this.props.value}

>

{this.props.label}

</div>

)

}

}

From the App, it receives a label, value, and onClick function. The Button component makes use of this data to render the button.

We haven’t talked about one component is the App, that’s Display.

The Display gets the state operations (Array) as a prop from the App.

class Display extends Component {

render() {

const string = this.props.data.join('')

return <div className="Display"> {string} </div>

}

}

The received data is operations from the state, it’s converted to the string to display in the Display component on top of the calculator. The operation you see on the display area is a string.

These were all the components you see on the screen.

**The operations in the React Calculator**

The operations are handled by the handleClick function, which is called upon clicking the button.

handleClick = e => {

const value = e.target.getAttribute('data-value')

switch (value) {

case 'clear':

this.setState({

operations: [],

})

break

case 'equal':

this.calculateOperations()

break

default:

const newOperations = update(this.state.operations, {

$push: [value],

})

this.setState({

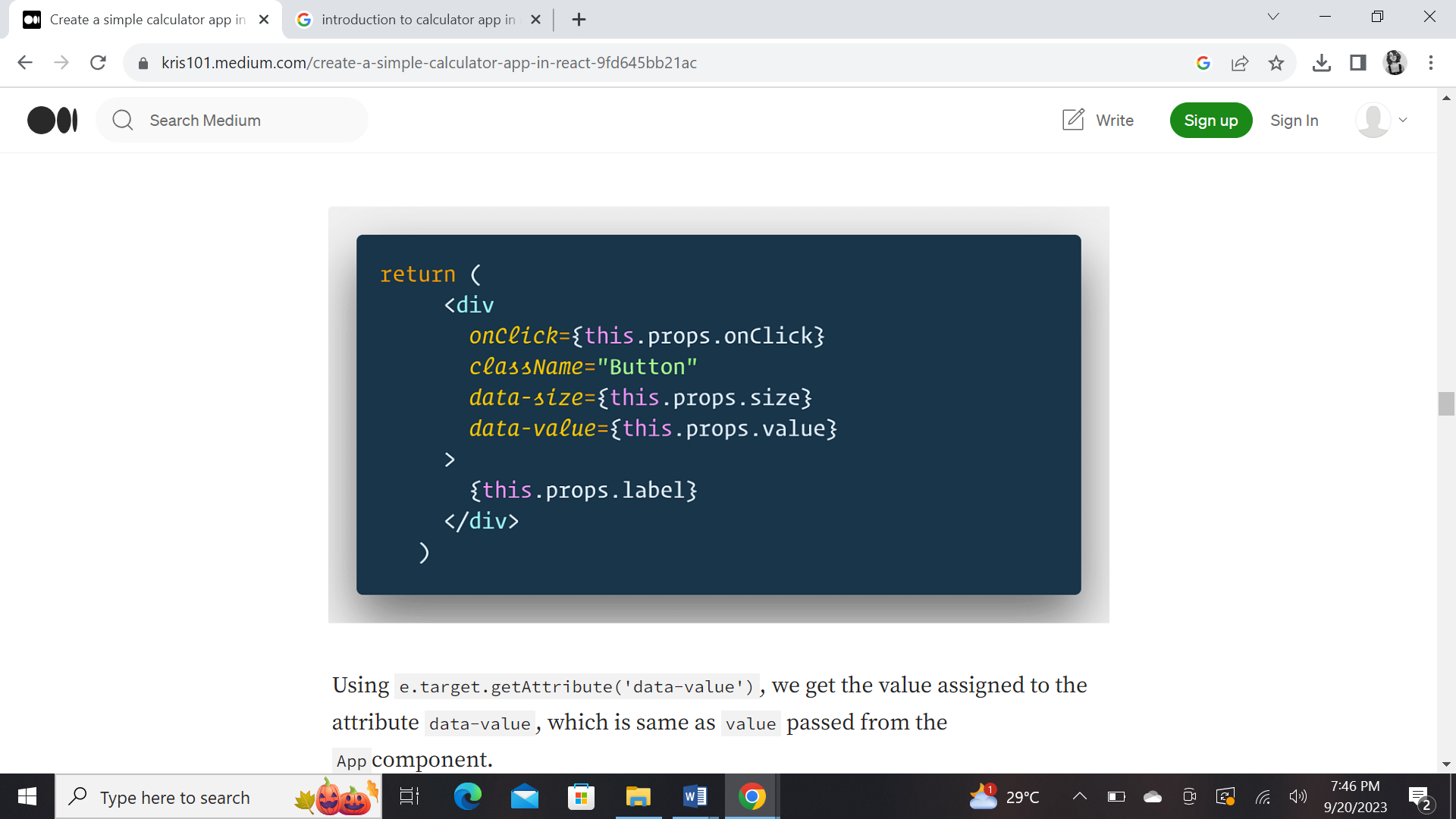
operations: newOperations,

})

break

}

Upon clicking the button, the event button is passed. If you look in the Button component, the value is stored as data-value.



Using e.target.getAttribute('data-value'), we get the value assigned to the attribute data-value, which is same as value passed from the App component.

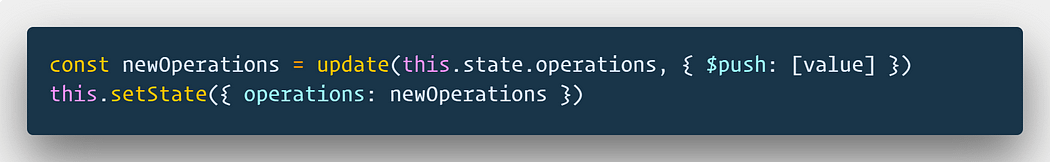
Using switch and case, we clear if the selected value was clear, that is clear button was pressed. The operations array is set to blank in the state.

The calculateOperations function executes on clicking equal, which we’ll talk about next. For all other options, we use an update function.

**Updating states in the react app**

Of the button is not clear, or equal it goes to the operations array. Which contains everything to calculate the final output on the display.

On the top of the App, we have imported update from immutability-helper, we’re using the same function.

Why update instead of directly pushing new value to the array?

this.state.comment = 'Hello' will not re-render the component.

this.setState({comment: 'Hello'}) will re-render the component.

It’s important that the components get re-render when the states are changed, that’s the fundamental concept of the react. You can not take care of all places the state is used and modify it one by one. We use the state instead of doing that.

The only place to assign this.state is constructor!

You can read more about this in the official documentation here.

What we do instead of modifying the operations array is to create a copy of it (newOperations) along with a new value pushed. Then we set the operations state to newOperations. This solves the problem and the functionality is provided by the library immutability-helper.

Once we update the state, react knows that the DOM needs update.

**Calculating the operations**

The calculator must calculate the answer of the operation!

calculateOperations = () => {

let result = this.state.operations.join('')

if (result) {

result = math.eval(result)

result = math.format(result, { precision: 14 })

result = String(result)

this.setState({

operations: [result],

})

}

}

It forms string out of an array.

If anything exists in the array, the string is evaluated with math.eval. We do not use the global eval function to evaluate this string, it’ll evaluate any JavaScript expression, not just a mathematical one. It’s a quite a bit of security issue. Instead of tinkering with the potential security issues, we use a package called mathjs.

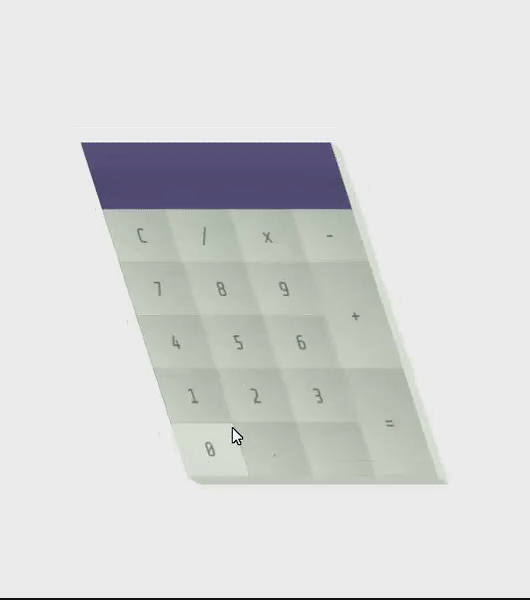
mathjs has it’s own eval function which only parses mathematical expressions.

math.eval() takes string and gives a number.

Why formatting?

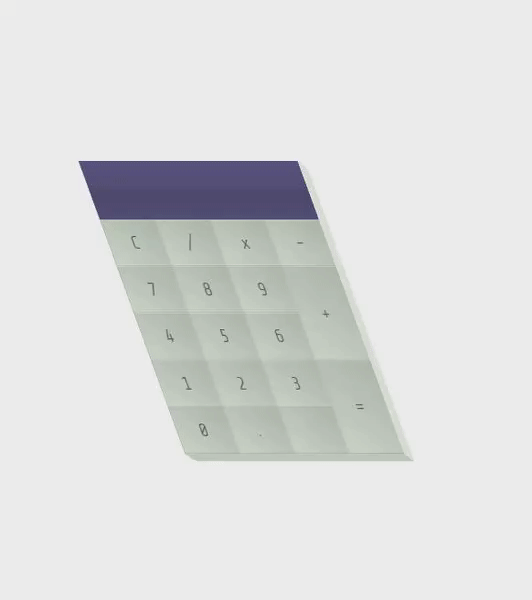
There’s a fundamental issue with the arithmetic in JavaScript. There’s only one type of number, unlike other languages. It has very high precision, it doesn’t care how many you’re actually using.

If I do not control the precision, here’s what I get…



Even with the 14 digit precision, we get the correct result. It’ll show correct result up to 14 decimal characters. That is enough for our case.

With the precision set I get this…



Then we set the state to the result so that the display gets updated.

A quick recap, the Display component on the top takes the state.operations to display input and result. This was the last component talked about above.

That is it, now you’ve a working calculator in react!

There are a few pitfalls though, the expression has to be valid. Something like 2\*/5 will crash the app. Expression 2.4.5 + 1 will also crash the app. This is a logical side (pure JavaScript logic) of the app and has nothing to do with the react.

It can be solved by having conditional statements in the handleClick before updating the operations with newOperations to check invalid operators (2\*/) or a decimal (.).