Hello World in React

```
import React from "react";
import ReactDom from "react-dom";

const element = <h1>Hello World</h1>;
// Babel will compile this call to React.createElement()
// So we have to import React even we direcly dont call it

console.log(element);
```

If we inspect and look at the console, we will get an object with *type react.element.*
Virtual Dom is a light-weight, in-memory representation of the UI. The object we see there is a part of *Virtual Dom.* Whenever the state of this object changes, react will get a new *react element* and compare with the previous one, figure out what is changed, reach out to the real *DOM* and update it accordingly.
So to render this inside of the real DOM,
`ReactDOM.render(element, document.getElementById("root"));`
The second argument is where in the real DOM, we wanna render this. In *index.html*, we have *div* with *id=root*, which is a container for our react application.

My first React Component

Add a new folder *components* inside *src* folder. Here we will put all our components. Inside component folder add a new file called *counter.jsx. Use .jsx* extension instead *.js* for better code completion. Now we use the extension *Simple React Snippets* for importing. Its a shortcut for repeated type of codes.
In *counter.jsx* create *counter* component as default export

```
import React, { Component } from "react";

class Counter extends Component {
    state = {};
    render() {
        return <h1>hello world!!!</h1>;
    }
}

export default Counter;
```

Import counter component in *index.js* file.

```
import Counter from "./components/counter";
```

Now we can use this as an XML element *<Counter />*

```
ReactDom.render(<Counter />, document.getElementById("root"));
```

Specifying Children

```
Render() {
    return <h1>hello world!!!</h1><button>Increment</button>;
}
```

This doesn’t work because JSX expressions must have one parent element. Because it gets call to *React.createElement(‘h1’).* But if we have two elements side to side then babel doesn’t know how to compile them into above expression. One solution is to wrap them with *<div>*.Now we have React Element *div* and inside we have two other elements *h1 and button*

```
render() {
    return (
        <div>
            <h1>hello world!!!</h1>
            <button>Increment</button>
        </div>
    );
}
```

Now we don’t want extra *div* that isn’t doing anything here. In this case we use *React.Fragment.* This is child of *React* object that we imported above.

```
<React.Fragment>
    <h1>hello world!!!</h1>
    <button>Increment</button>
</React.Fragment>
```

Embedding Expressions

Now we want to add value dynamically in component which is done by *state*. State is an object that has any data a *component* needs

```
state = {
    count: 0
};
render() {
    return (
        <React.Fragment>
            <span>{this.state.count}</span>
            <button>Increment</button>
        </React.Fragment>
    );
}
```

Inside curly braces we can add any valid Javascript expressions. Expression is something that produces a value.

```
class Counter extends Component {
    state = {
        count: 1
    };
    render() {
        return (
            <React.Fragment>
                <span>{this.formatCount()}</span>
                <button>Increment</button>
            </React.Fragment>
        );
    }

    formatCount() {
        const { count } = this.state; //Object destructuring
        return count === 0 ? <h2>Zero</h2> : count;
                                    //We can use JSX expressions
    }
}
```

JSX expressions are just like normal Javascript Objects. We can use them as Variables, Arguments, pass to and return from a function etc.

Setting Attributes

```
state = {
    count: 0,
    imageUrl: "https://picsum.photos/200"
};
```

Inside return:

```
<img src={this.state.imageUrl} alt="" />
```

Now lets use CSS styles as dynamic Attributes

```
styles = {
    fontSize: 50,
    fontWeight: "bold"
};

render() {
    return (
        <React.Fragment>
            <img src={this.state.imageUrl} alt="" />
            <span style={this.styles} className="badge badge-primary mx-2">
                {this.formatCount()}
            </span>
            <button className="btn btn-success mx-2">Increment</button>
        </React.Fragment>
    );
}
```

Or we can Style elements inside the tag

```
<span
    style={{ fontSize: 30, color: "red" }}
    className="badge badge-primary mx-2"
>
```

Rendering Classes Dynamically

Lets make this one of the functions

```
getBadgeClasses() {
    let classes = "badge m-2 badge-";
    classes += this.state.count === 0 ? "warning" : "primary";
    return classes;
}
```

Now we can modify class dynamically as:

```
<span style={this.styles} className={this.getBadgeClasses()}>
```

Rendering Lists

Map function in ES6 allows us to wrap an Array items into tags and whatever we want

```
state = {
    count: 0,
    imageUrl: "https://picsum.photos/200",
    tags: ["tag1", "tag2", "tag3"]
};
```

```
<ul>
    {this.state.tags.map(tag => (
        <li key={tag}>{tag}</li>
    ))}
</ul>
```

Conditional Rendering

If we have minimum one item on the list we will display otherwise we wanna display message like *there are no tags* etc. We don't have if else like conditionals in JSX to render elements conditionally we have to go back to plain Javascript. We will use *renderTags()* method to implement conditionals

```
renderTags() {
    if (this.state.tags.length === 0) {
        return <p className="text-warning">There are no tags</p>;
    } else {
        return (
            <ul>
                {this.state.tags.map(tag => (
                    <li key={tag}>{tag}</li>
                ))}
            </ul>
        );
    }
}
```

Now we implement *&&* for conditional rendering. Inside *render()*

```
<div>
    <p>{this.state.tags.length === 0 && "Please create a new tag"}</p>
    {this.renderTags()}
</div>
```

This expression in ES6 returns the second argument if the first argument is true.

Handeling Events

Events like onClick, keyUp, keyDown etc

```
<button onClick={this.handleIncrement} className="btn btn-success mx-2">
    Increment
</button>
```

Here we are not calling the function like *this.handleIncrement()* but we are passing reference to it. This is different than handling events in pure JavaScript. Now lets increase *count* when clicked

```
handleIncrement() {
    console.log("Increment Clicked", this.state.count);
}
```

But this doesn't work. When we click, our console returns *this* as *undefined.* Currently we don't have access to *state* property.

In JS, depending upon how a function is called *this* behaves differently. If a function is called as a method of an object *this* always returns reference to that object. But if a function is called as standalone function *this* by default will return window object which returns undefined if strict mode is enabled.

So we don't have access to *this* in *onClick* event handler. Solution to this we talked about in earlier topic called *bind* method. Lets add a constructor:

```
constructor() {
    console.log("Constructor", this);
}
```

This will create error: *Syntax error: 'this' is not allowed before super()*

So first we have to call constructor of the parent class using ***super*** keyword.

Now in constructor, we have access to *Counter* object(Class is also object in JS). And *this* returns *Counter* object

```
constructor() {
    super();
    this.handleIncrement = this.handleIncrement.bind(this);
}
```

Functions in JS are objects so they have properties and methods and *bind()* is one of them. With this method we can set the value of *this*. This *bind()* method will return new instance of *handleIncrement()* function and in that function *this* is always referencing the current object which is *Counter. So*

`this.handleIncrement.bind(this);`

returns new function where we can get that function and reset *handleIncrement()* like

`this.handleIncrement = this.handleIncrement.bind(this);`

Now when we click *Increment* button we no longer see *undefined.* With this we will be able to update the *state* property. The above procedure is one solution which is noisy as hell. There is another way to solve this problem which is currently experimental and likely to break in the future.

Instead of adding a constructor we can convert this function into an arrow function. Arrow function don't rebind *this* keyword but in inherits it.

```
handleIncrement = () => {
    console.log("Increment Clicked", this.state.count);
};
```

This works without the need of Constructor and bind method. Now we are ready to update the state.

Updating the State

Now we have access to the current object. In react we don't modify the state directly like

```
handleIncrement = () => {
    this.state.count++;
};
```

This isn't gonna work. Now when we click Increment button technically *count* is incremented but react is not aware of that. It's not updating the view. To solve this problem we use one of the method we inherit from the base *component* class ***this.setstate()***

This method tells react that we are updating the state then it will figure out what part of the state is changed and based on that it will bring the DOM in sync with the *virutalDOM*

```
handleIncrement = () => {
    this.setState({ count: this.state.count + 1 });
};
```

({…….}) this syntax because we are changing one of the properties of an Object.

What happens when State changes?

When we click the increment button, *this.setState()* is telling React that the state of this component is going to be changed. React will then schedule a call to the *render* method. This is an asynchronous call.

React will compare Old and New react element side by side and detect the changes. In this case

```
<span style={this.styles} className={this.getBadgeClasses()}>
    {this.formatCount()}
</span>
```

Nothing but only *span* element is changed. We can get a preview in using inspect in browser, click *Increment* and see.

Passing Event Arguments

In real world applications we need to pass arguments in our events. Lets pass a specific product when we click *increment*. We cannot pass argument on *onClick* like

```
<button
    onClick={this.handleIncrement(product)}
    className="btn btn-success mx-2"
>
```

because we should only pass function reference.

One solution is temporarily defining another method and pass an argument and pass this function reference on *onClick* event.

```
handleIncrement = product => {
    console.log(product);
    this.setState({ count: this.state.count + 1 });
};
```

```
doHandleIncrement = () => {
    this.handleIncrement({ id: 1 });
};
```

In render method:

```
<button
    onClick={this.doHandleIncrement}
    className="btn btn-success mx-2"
>
```

Now when we click *increment* we will see product argument (*id: 1*) in console. But this way of writing code is messy. Better solution is to use an inline function.

```
onClick={() => this.handleIncrement({ id: 1 })}
```

When rendering list of products in a shopping cart, in our map method, we have access to our product object. Instead of hardcoding object, we will pass a reference to the product that we are currently rendering. Now we no longer need *doHandleIncrement*

```
product = {
id: 1,
name: "Soap"
};
```

```
onClick={() => this.handleIncrement(this.product)}
```

Summary

In above section we learned the essence of React Components. We learned

- JSX
- Rendering Lists
- Conditional Rendering
- Handling Events
- Updating the State of Components

In next section we will look at composing components to build complex user interfaces.

Composing Components

Lets make a *Counters* component where we will render *Counter* component we previously built and in *index.js* we will render *Counters* component instead of *Counter* component.

In *components* folder add new file called *counters.jsx*

```
import React, { Component } from "react";
import Counter from "./counter";
```

```
class Counters extends Component {
     state = {};
     render() {
          return (
               <div>
                    <Counter />
                    <Counter />
                    <Counter />
               </div>
          );
     }
}
export default Counters;
```

Now each component have their own state completely isolated from other components. Instead of hard-coding counter components lets take an array of counter objects to our state property and render them using map method

```
state = {
     counters: [
          { id: 1, value: 0 },
          { id: 2, value: 0 },
          { id: 3, value: 0 },
          { id: 4, value: 0 },
     ]
};
```

```
render() {
     return (
          <div>
               {this.state.counters.map(counter => (
                    <Counter key={counter.id} />
               ))}
          </div>
     );
}
```

Passing Data to Components

There are props which we pass as an argument in Components. They can be accessed by child component by *this.props.dataWeWant* .

In *Counters* component:

```
<Counter key={counter.id} value={counter.value} selected />
```

here *value* and *selected* are props. We can write *selected=true* but it is true by default. *key* is not props because it is special argument of a component.

In *Counter* component:

```
state = {
    count: this.props.value,
```

Passing Children

We have a special prop called children and we use this when we want to pass something between opening and closing tag of components.

```
<Counter key={counter.id} value={counter.value} selected>
    <h4>This is a demo</h4>
</Counter>
```

Now when we console log props in *Counter* component and inspect we will see new properties called *children* which is React Element and it's type is *h4.* That meant we can render children wherever we want. Now in *Counter:*

```
render() {
    console.log(this.props);
    return (
        <React.Fragment>
            {/* {Rendering Children(Contents inside tags) dynamically} */}
            {this.props.children}
            <img src={this.state.imageUrl} alt="" />
            <span style={this.styles} className={this.getBadgeClasses()}>
                {this.formatCount()}
            </span>
```

Now each counter component has a title. Instead of hard-coding a title we can have something like:

```
<Counter key={counter.id} value={counter.value} selected>
    <h1>Counter {counter.id}</h1>
</Counter>
```

Now we will have heading like Counter 1, Counter2, etc.

Props vs State

Props include data that we give to a component whereas State includes data that is local to a component. Props is read-only. We cannot change it inside components.

Raising and Handling Events

Lets add a delete button next to each counter and add onClick event *handleDelete()* but if we look at the state we only have value property. In order to delete it we need to delete from array of counters from *Counters* component. This brings to very important rule of thumb in react:

The component that owns a piece of the state, should be the one modifying it.

So modifying the state should be done by *Counters*. We will be raising event from *Counter* component and handling it from *Counters* component. On *Counters:*

```
<Counter
    id={counter.id}
    onDelete={this.handleDelete}
    value={counter.value}
    selected
>
```

On *Counter:*

```
<button onClick={this.props.onDelete} className="btn btn-danger m-2">
    Delete
</button>
```

Updating the State

On *handleDelete* function we have an argument *counterId* to figure which counter component to delete. In *Counters:*

```
handleDelete = counterId => {
    console.log("Event handler called!!", counterId);
};
```

In *Counter:*

```
<button
    onClick={() => this.props.onDelete(this.props.id)}
    className="btn btn-danger m-2"
>
```

Now when we click *delete* button console will return *Event handler called!! 1*, *Event handler called!! 2* according to which component button we clicked. In React, we don't update the state directly or we don't delete items in *counters* array. Instead we will create new array without the given *counter* and then call the *setState* method of our component and let React update the state.

```
handleDelete = counterId => {
    const counters = this.state.counters.filter(c => c.id !== counterId);
    this.setState({ counters: counters });
};
```

In *setState* method, key and value are same so we can simplify like:

```
this.setState({ counters });
```

Now look at this:

```
<Counter
    id={counter.id}
    onDelete={this.handleDelete}
    value={counter.value}
    selected
>
```

We are hard-coding multiple properties. Both *value* and *id* are properties of *counter* objects. So we can pass *counter* object as a single argument and access these properties from child component.

```
<Counter
    onDelete={this.handleDelete}
    counter={counter}
    selected
>
```

Now we make changes on *onClick* button by adding prefix *counter*

```
<button
    onClick={() => this.props.onDelete(this.props.counter.id)}
    className="btn btn-danger m-2"
>
```

In state also:

```
state = {
    count: this.props.counter.value,
```

Single source of Truth

Lets we wanna add button which resets all the counters. In *Counters*

```
handleReset = () => {
    const counters = this.state.counters.map(c => {
        c.value = 0;
        return c;
    });
    this.setState({ counters });
};
```

Back in the browser when we click *Reset* there is no change. The reason is we don't have single source of truth. Open developer tools and go to React menu. Click *Counters* component at right side we can see

```
counters:
Array[4]
```

We open these and find value is 0. We successfully updated the state but we are not seeing the changes in the DOM. Expand *Counters* and click last *Counter* and inside props value is 0 but in state value is 3. Each of our component has their own local state and their value is disconnected from *counter* object

```
state = {
    count: this.props.counter.value,
```

We are initializing *count* property of our state object based on what we get from our props. It gets executed only once when an instance of counter component is created and it becomes local state. When we click *Reset* button the local state of *counter* component is not updated.

Removing the Local State

Now we rely only on props to receive the data we need. This is called controlled component. A controlled component doesn’t have it’s own local state and raises events whenever a data needs to be changed. First delete the state property. Find any reference to *this.state* and update them accordingly. h*andleIncrement* should be in parent component since it’s a controlled component. In the increment button:

```
<button
    onClick={() => this.props.onIncrement(this.props.counter)}
    className="btn btn-success mx-2"
>
```

Replace all *this.state* with *this.props.counter*

```
getBadgeClasses() {
    let classes = "badge m-2 badge-";
    classes += this.props.counter.value === 0 ? "warning" : "primary";
    return classes;
}
```

Now *handleIncrement* is in *Counters* component. We will update state but not directly like:
counters[0].value++;

This is a direct no-no in React. We will first clone that object and update it’s state.

```
handleIncrement = counter => {
    const counters = [...this.state.counters];
    const index = counters.indexOf(counter);
    counters[index] = { ...counter };
    counters[index].value++;
    this.setState({ counters });
};
```

Multiple Components in Sync

Lets make a root component *App* which contains *Counters* and *Nav* which is navigation bar. Add new files under component folder *app.jsx* and *navbar.jsx*.

```
class App extends Component {
    state = {};
    render() {
        return (
            <React.Fragment>
                <NavBar />
                <main className="container">
                    <Counters />
                </main>
            </React.Fragment>
        );
    }
}
```

There is no parent-child relationship between *Counters* and *NavBar.* So how can we display total no of counters on our Navigation Bar? We are lifting the state up to it’s parent *App* so we can pass it to all the children using props.

Lifting State Up

In *Counters*, we have all the states as well as the methods to modify or mutate the state. We should move all of them to it's parent component *App* and pass it down to *Counters* using props. Now we have all state and method in *App* we will pass props to *Counter* component:

```
<React.Fragment>
    <NavBar
        totalCounters={this.state.counters.filter(c => c.value > 0).length}
    />
    <main className="container">
        <Counters
            counters={this.state.counters}
            onReset={this.handleReset}
            onIncrement={this.handleIncrement}
            onDelete={this.handleDelete}
        />
    </main>
</React.Fragment>
```

Now in *Counters* we will change all the function reference into *this.props.* For example

```
<button onClick={this.props.onReset} className="btn-primary btn-sm m2">
    Reset
</button>
```

```
<Counter
    onDelete={this.props.onDelete}
    onIncrement={this.props.onIncrement}
    counter={counter}
    selected
>
```

We have already passed a prop on *NavBar* for total no of counter component having value greater than one. Lets get that from *NavBar*

```
class NavBar extends Component {
    render() {
        return (
            <nav className="navbar navbar-light bg-light">
                <a className="navbar-brand" href="#">
                    Navbar
                    <span className="badge badge-pill badge-secondary m-2">
                        {this.props.totalCounters}
                    </span>
                </a>
            </nav>
        );
    }
}
```

Stateless Functional Components

NavBar component don’t have a state, event-handlers and have single method called *render &* we are getting all data through props. Now we can convert these type of components into *Stateless Functional Component.*

```
const NavBar = props => {
    return (
        <nav className="navbar navbar-light bg-light">
            <a className="navbar-brand" href="#">
                Navbar
                <span className="badge badge-pill badge-secondary m-2">
                    {props.totalCounters}
                </span>
            </a>
    </nav>
    );
};

export default NavBar;
```

Destructuring Arguments

While working with complex application we might have multiple references to *props.*

```
class Counters extends Component {
    render() {
        const { onReset, onDelete, onIncrement, counters } = this.props;
        return (
            <div>
                <button onClick={onReset} className="btn-primary btn-sm m2">
                    Reset
                </button>
                {counters.map(counter => (
                    <Counter
                        onDelete={onDelete}
                        onIncrement={onIncrement}
                        counter={counter}
                        selected
                    >
                        <h1>Counter {counters.indexOf(counter) + 1}</h1>
                    </Counter>
                ))}
```

Lifecycle Hooks

Our components goes through a few phases during their lifecycle. The first phase is the **mounting** phase which is when a instance of a component is created and inserted into the dom. There are method called *lifecycle hooks* that allows us to hook into certain moments during the lifecycle of a component and do something. In mounting phase we have 3 lifecycle hooks **constructor, render** and **componentDidMount.** Second lifecycle phase is the **update** phase & happens when the state or props of the component gets changed. In these phase we have two lifecycle hooks **render** & **componentDidUpdate**. The last phase is the unmounting phase which is when the component is removed from the dom such as when we delete the counter. This have one lifecycle hook **componentWillUnmount**.

Mounting Phase

ComponentDidMount(): This method is called after the component is rendered into the dom & is perfect place to call **Ajax** to get the data from the server.
The order in which the methods in mounting phase are :
1: Constructror, 2: Render, 3: Mount
When a component is rendered, all it's children are also rendered recursively. The order is the order we declare them in our code. 1st child renders and it's children renders going down to the base of hierarchy and only 2nd child renders.
Note: We can only use lifecycle hooks in class components, not in functional components.

Updating Phase

This phase happens whenever the props or state of component changes.

```
class Counter extends Component {
    componentDidUpdate(prevProps, prevState) {
    console.log("prevProps", prevProps);
    console.log("prevState", prevState);
```

```
    if (prevProps !== prevState) {
        //Ajax call and get new data from the server.
    }
}
```

Unmounting Phase

componentWillUnmount() is called just before a component is removed from the DOM

```
componentWillUnmount() {
    console.log("Component Unmounted");
}
```

This will be perfect opportunities for doing cleanups.