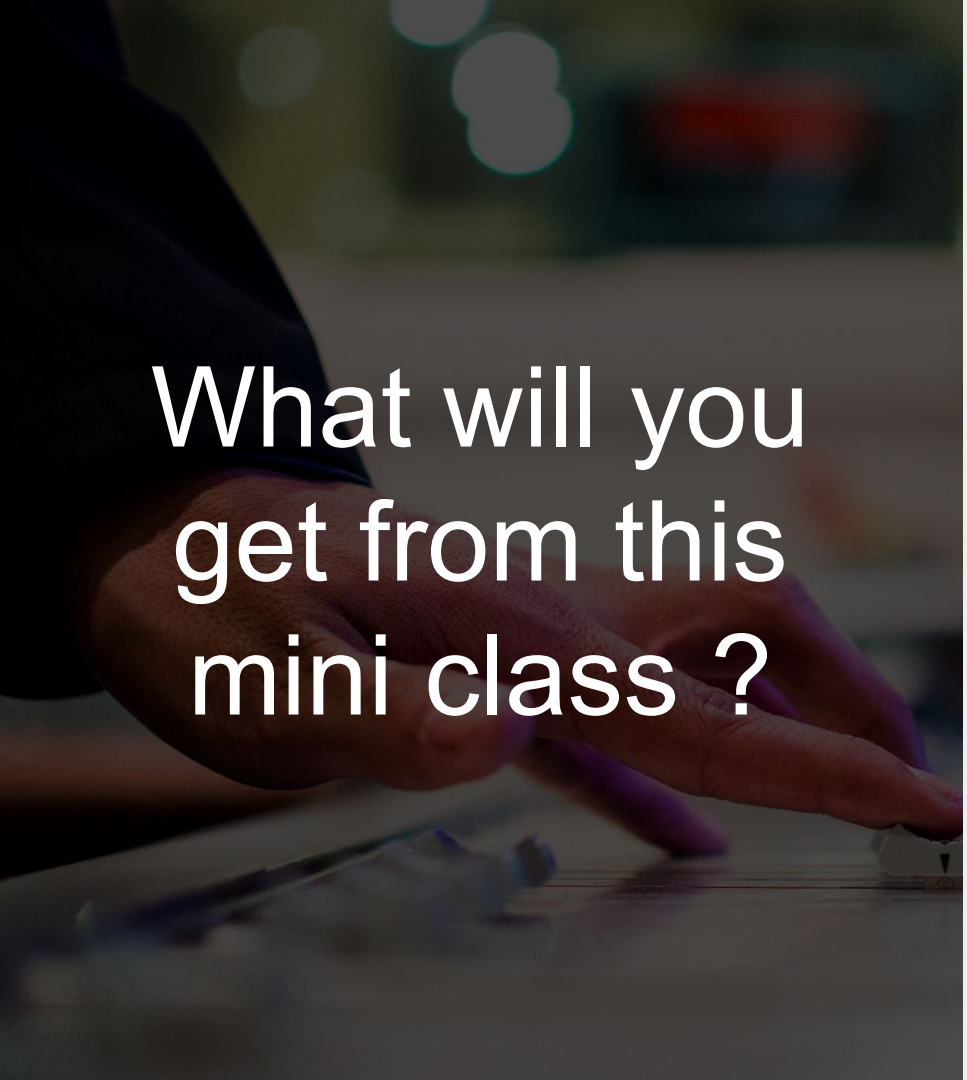
React Mini Class

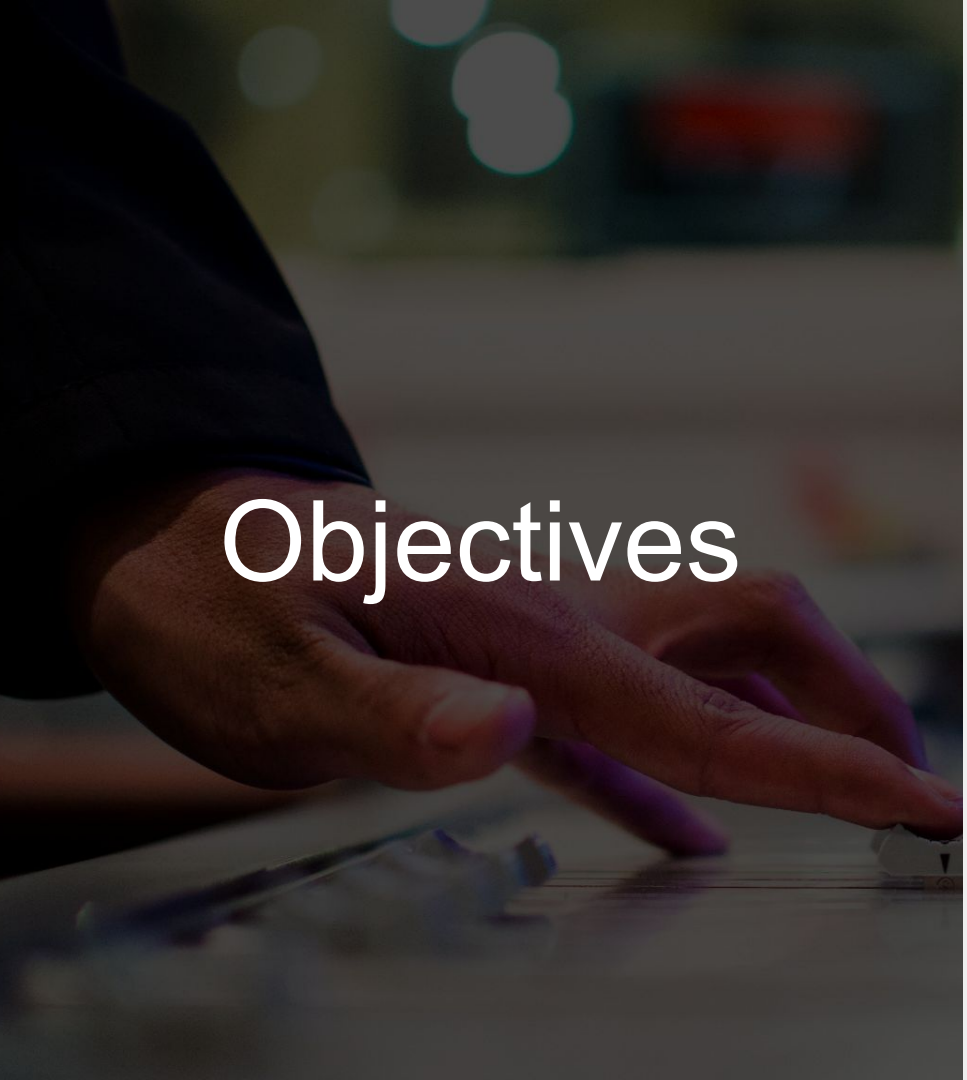
A crash course on ReactJS

- M. Yauri Attamimi



What will you get from this mini class ?

- Brief introduction to React
- Articulate the React Architecture
- Build simple React application



Objectives

By the end of this workshop, you will be able to:

- Master React Fundamentals
- Build Reusable Components
- Render Data with React
- Handle Events
- Debug your React Apps

About the Speaker

M. Yauri Maulana Attamimi

- 15 years (or so) in Software Development
- LOTS of project experience
- Java, NodeJS, Golang
- Microservices Provocateur
- AI & Blockchain Enthusiast
- TDD and Clean Code Evangelist
- VP Engineering & Co-Founder of Automate.id

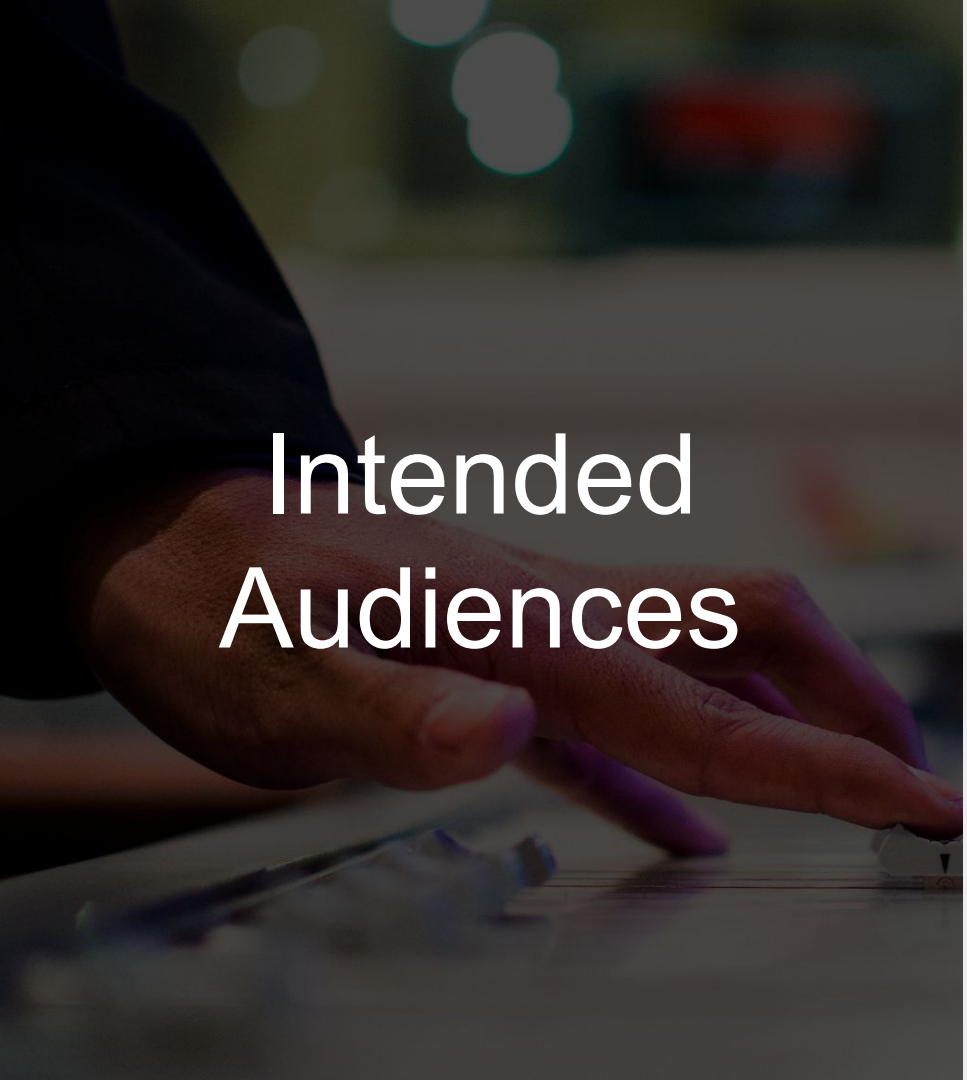
https://about.me/yauritux

My Professional Mission :

Guiding individuals and organizations to commercial success through the application of modern technologies

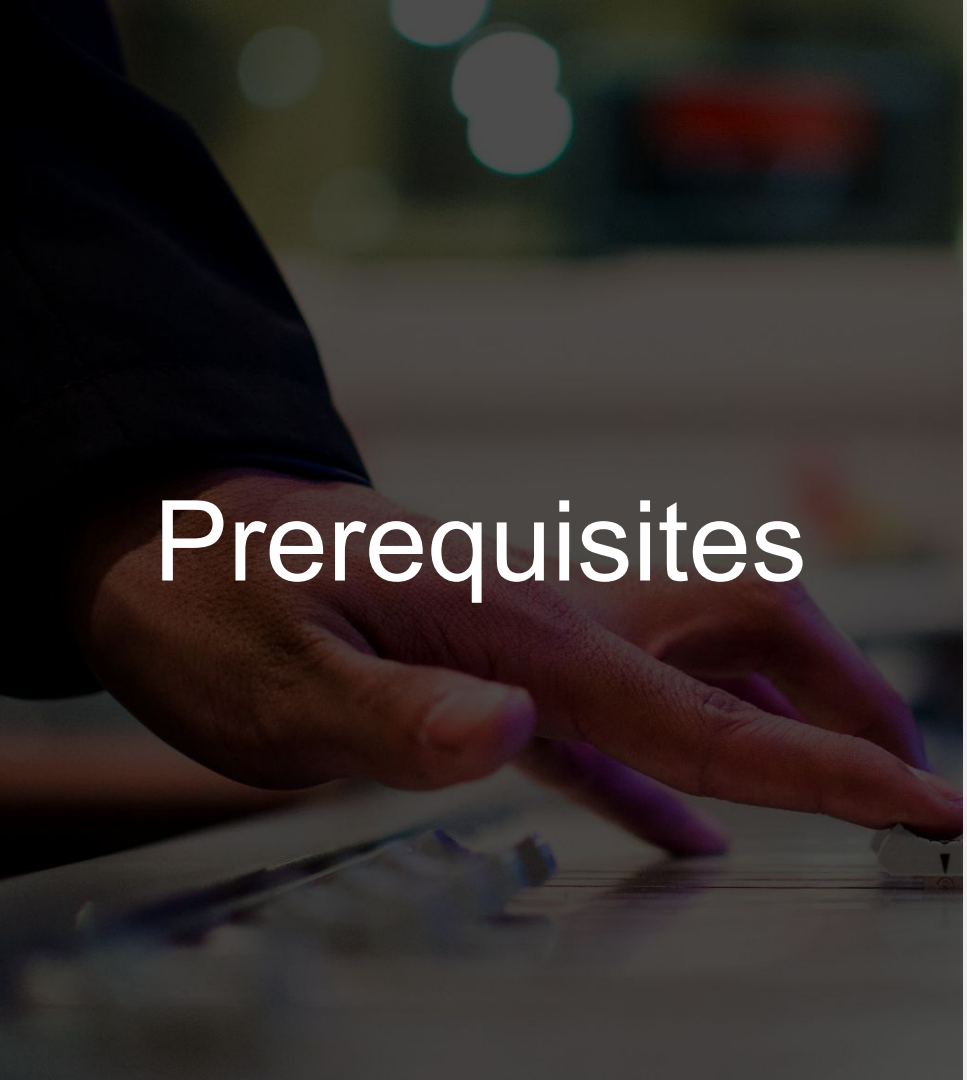
@yauritux

DZone https://dzone.com/users/366249/yauritux.html



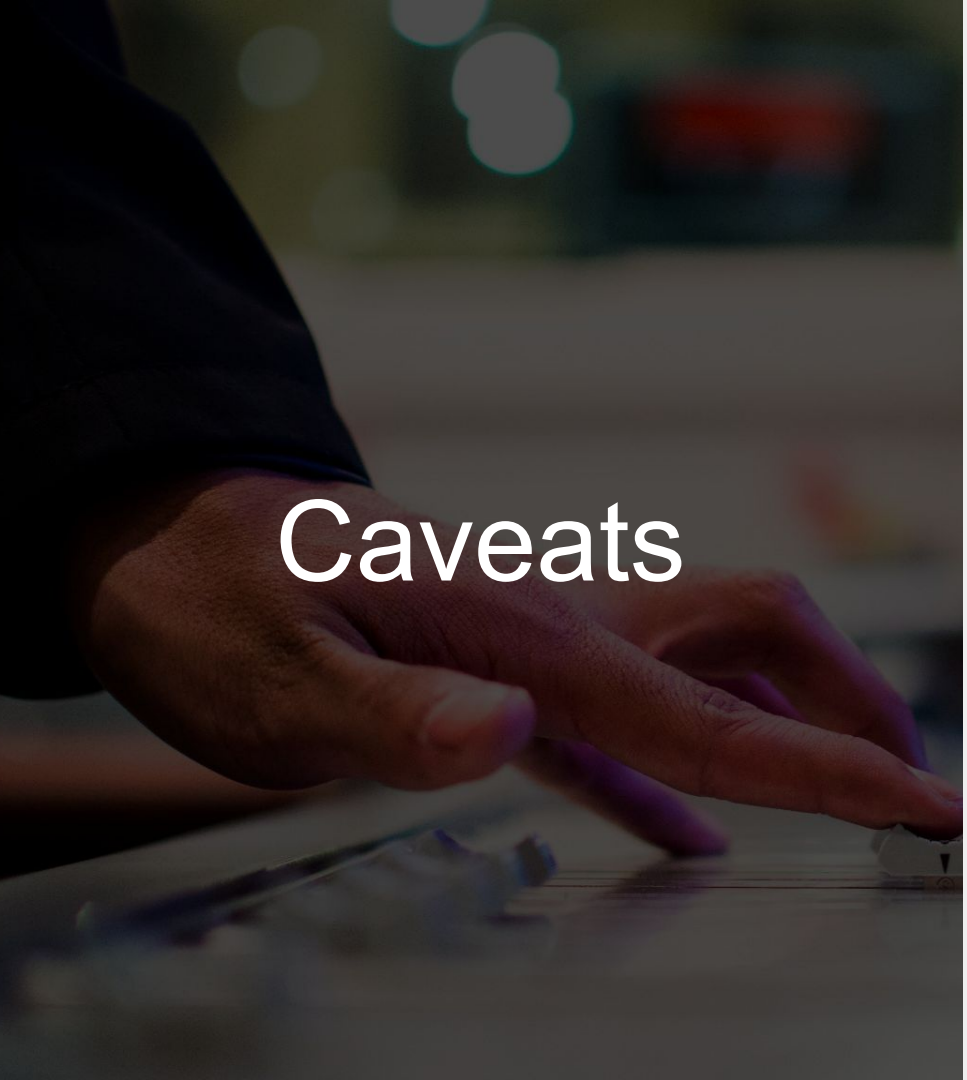
Intended Audiences

- Someone with no prior knowledge on React, but has little knowledge about web development stuffs
- Someone with basic knowledge of JavaScript



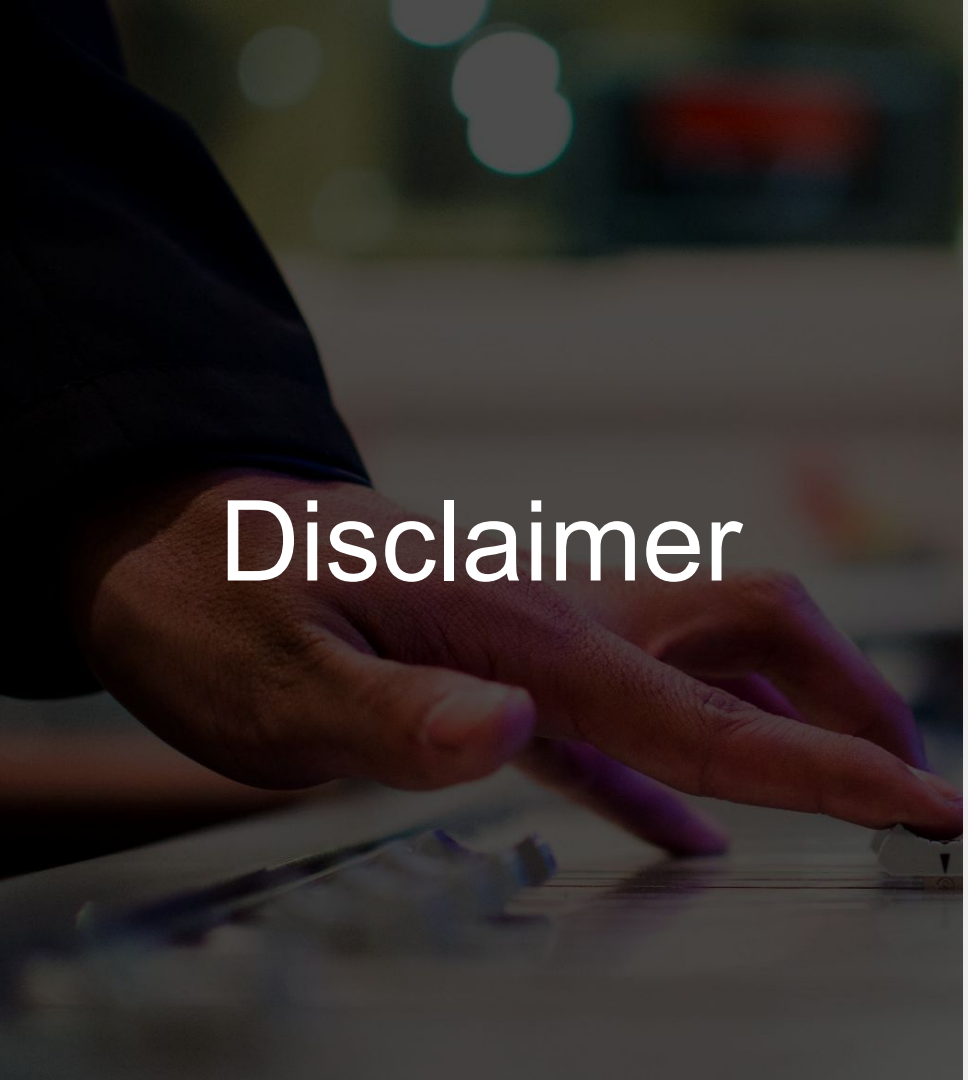
Prerequisites

- Basic Knowledge of HTML, CSS, and JavaScript.
- Basic Understanding of ES6 Features. You should at least know the following features:
  - let, const
  - arrow functions
  - imports and exports
  - classes
- Basic understanding on how to use `npm`
- Basic knowledge of Web Technologies (Web protocols, etc)
- Git - basic knowledge



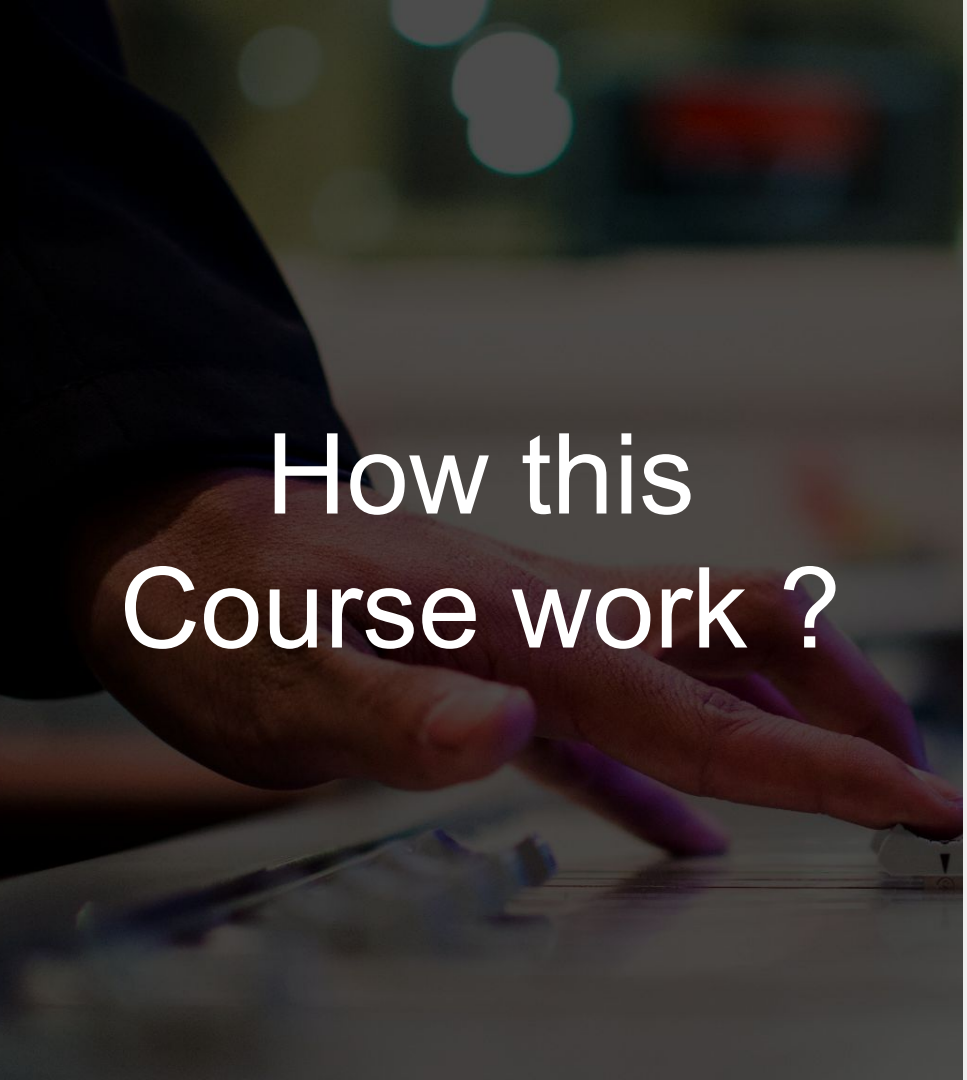
Caveats

- Unlikely to cover all topics within short hours
  - Goal is to be reasonably comprehensive
  - Enable you to develop your own React applications
  - Enable you to fill in gaps yourself once you know what does it need to be a master



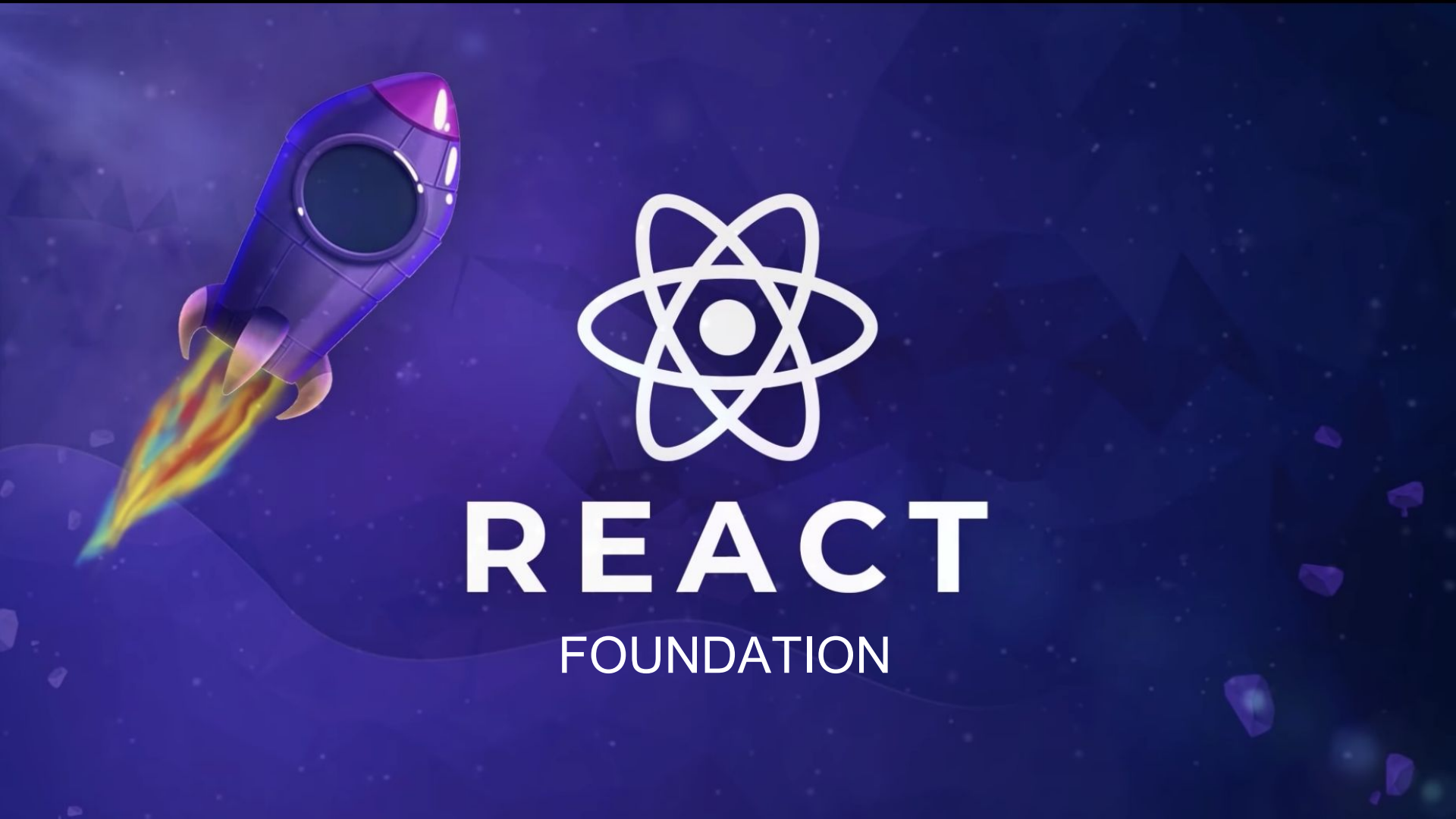
Disclaimer

The information provided here is designed to provide helpful information on the subjects discussed and just my own opinion based on my proven experiences (not represent any entities)



How this Course work ?

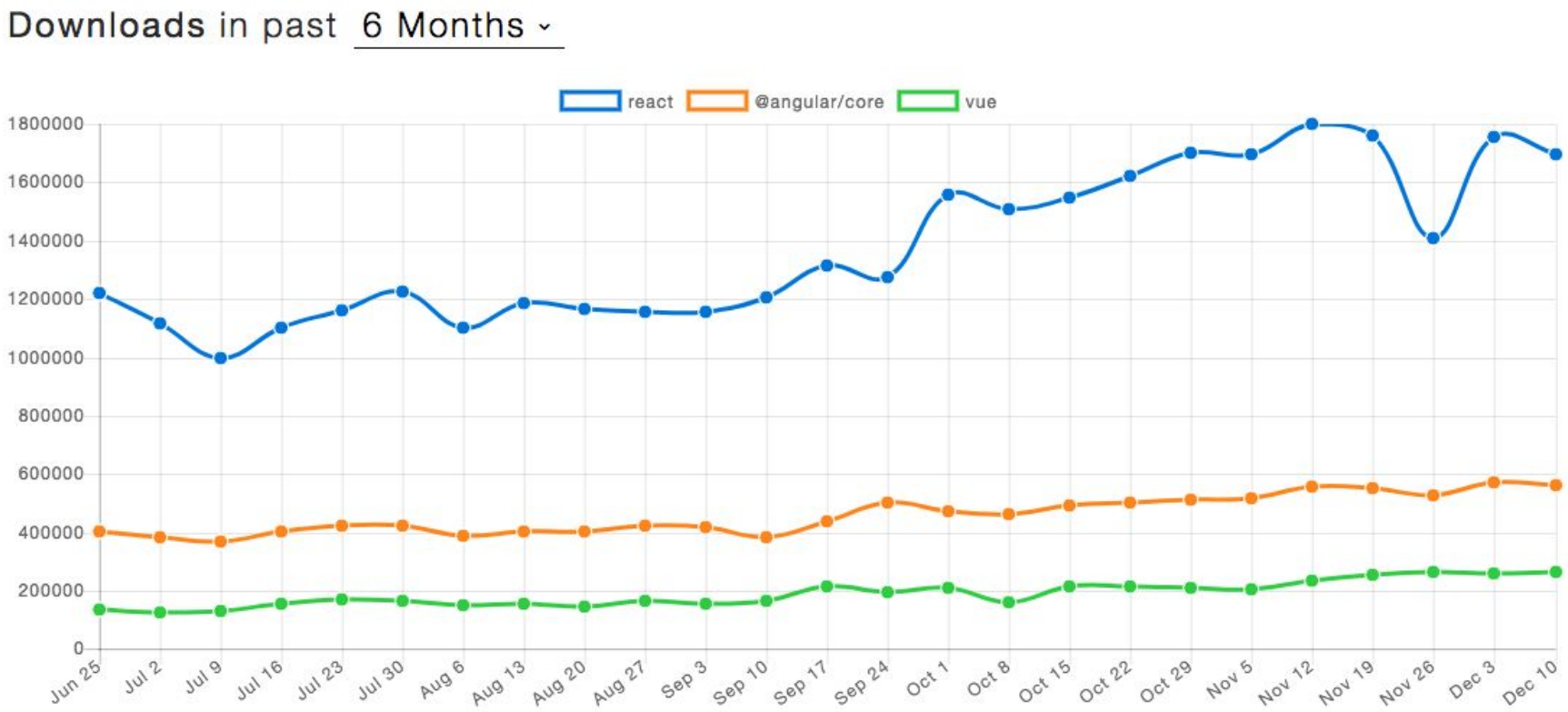
- Slide to explain concepts
- Exercises to reinforce concepts
- IDE recommended : Visual Studio Code


REACT
FOUNDATION

What is React ?

- React is a JavaScript library for building fast and interactive user interfaces.
- React was developed on Facebook in 2011 and currently is the most popular JavaScript library for building interfaces

React Trends (Google)



React Component (I)

- Component is the heart of all React applications
- Component essentially is a piece of the user interface
- When building a React app, we build a bunch of independent, isolated, and reusable components…, then compose them to build complex user interfaces.
- Every React application has at least one component which we refer to as the “Root” component.
- “Root” component represents the entire application and contains other children components.
- Every React application essentially is a tree of components.

React Component (II) - Example

Comprises of these following components:

- Navbar
- Profile
- Trends
- Feed
    - Tweet
        - Like

And so forth.

As we can see, each component is a piece of UI.

React Component (III) - Continued

- In term of implementations, a component is typically implemented as a JavaScript class which has some **states** and **render** method.

```
class Tweet {
    state = {};
    render() {
    }
}
```

- The `state` is the data that we want to display when the component is rendered.
- The `render` method is responsible to describing what the UI should be looked like.
- The output of the `render` method is the **React element** whish is a simple plain JavaScript object that maps with the **DOM element** (represents the **DOM element** in the memory a.k.a **Virtual DOM**)

Virtual DOM

- Unlike the browser with the real DOM, Virtual DOM is cheap to create.
- When we change the state of the component, we get a new React element. React will then compare this element, and its children to the previous one and it figures out what has changed, then it will update the part of the real DOM and keep it sync with the Virtual DOM.
- We don’t need to write any code directly manipulate the DOM or even attach an event handler to the DOM element. We simply change the state of the component (Virtual DOM), and React will automatically update the DOM to match that state.

React vs Angular ?

- Both is similar in terms of Component-Based Architecture.
- Angular is a framework with complete solution, while React is a frontend (view) library.
- React is much simpler than angular (i.e. short learning curve)

Setup

1. Download and install the latest stable version of NodeJS
2. Install React toolkit globally

   ```
   npm install -g create-react-app@1.5.2
   ```

3. Install Code Editor (e.g. Visual Studio Code / VSCode)
4. Install these 2 extensions within your installed VSCode:
   a. **Simple React Snippets** → developed by Burke Holland
   b. **Prettier** → developed by Esben Petersen
5. Settings to trigger prettier on file saved.
   a. **Accessing menu Code > Preferences > Settings**
   b. Under **User Settings** tab, add a new pair of key-values: `"editor.formatOnSave": true`

Our First React App

1. Open terminal.
2. Execute command: create-react-app <app_name>. E.g. :

   ```
   create-react-app hello-react-app
   ```

3. Go to the created folder (e.g. `hello-react-app`).
4. Run the program.

   ```
   npm start
   ```

5. Pay attention to the generated project skeleton.
6. Change the display with your own custom JSX (e.g. trying to display “Hello React” on the browser page)

Our First React Component

1. Create another project from your terminal (lets name it: `counter-app`)
2. Go to the created folder (i.e. `cd counter-app`)
3. Test to ensure it's running (i.e. `npm run`)
4. Add `bootstrap` to the `counter-app` project.

   ```
   npm i -S bootstrap@4.1.1
   ```

5. Open up `index.js` file.
6. Import bootstrap

   ```
   import 'bootstrap/dist/css/bootstrap.css';
   ```

Our First React Component - Continued

1. Create another folder within the `src` folder (name it "`components`").
2. Add a new file named **counter.jsx** into the `components` folder that just created.
3. Write these following code inside the **counter.jsx** file (use the shortcut provided by "Simple React Snippets" extension to help).

```
import React, { Component } from "react";
class Counter extends Component {
    render() {
        return <h1>Hello React</h1>;
    }
}
export default Counter;
```

Our First React Component - Continued

1. Open the index.js file, and write the following code:

```
…..
import Counter from ‘./components/counter’;
…..
ReactDOM.render(<Counter />, document.getElementById(‘root’));
registerServiceWorker();
```

2. Run the application again and check what being displayed on your browser.

Embedding Expressions

1. Open the counter.jsx file, and add the increment button (**note:** don't forget to wrap within a div or another react fragment since JSX will be compiled into React.createElement which is supposed to receive only one element).

```
…...
return (
    <React.Fragment>
        <h1>Hello React</h1>
        <button>Increment</button>
    </React.Fragment>
);
…...
```

2. Test the app again

Embedding Expressions - Continued

```
import React, { Component } from “react”;
export default class Counter extends Component {
    state = { count: 0 };
    render() {
        return (
            <React.Fragment>
                <span>{this.formatCount()}</span>
                <button>Increment</button>
            </React.Fragment>
        );
    }
    formatCount() {
        let {count} = this.state;
        return count === 0 ? “Zero” : count;
    }
}
```

Setting Attributes

```
import React, { Component } from “react”;
export default class Counter extends Component {
     state = {
          count: 0,
          imageUrl:’https://laracasts.com/images/series/circles/do-you-react.png’
     };
     render() {
          return (
               <React.Fragment>
                    <img src={this.state.imageUrl} alt=”” width=”25” height=”25”/>
                    <span>{this.formatCount()}</span>
                    <button>Increment</button>
               </React.Fragment>
          );
     }

     ...
}
```

Applying Class Style

```
...
export default class Counter extends Component {
    state = { count: 0 };
    render() {
        return (
            <React.Fragment>
                <span className="badge badge-primary m-2">
                    {this.formatCount()}
                </span>
                <button className="btn btn-secondary btn-sm">
                    Increment
                </button>
            </React.Fragment>
        );
    }
    ...
```

Applying CSS Style

```
...
    styles = { fontSize: 10, fontWeight: 'bold' };
    render() {
        return (
            <React.Fragment>
                <span style={this.styles} className="...">
                    {this.formatCount()}
                </span>
                <button className="btn btn-secondary btn-sm">
                    Increment
                </button>
            </React.Fragment>
        );
    }
    ...
```

Applying CSS Style - Another Way

```
...
    render() {
        return (
            <React.Fragment>
                <span style={{fontSize:10}} className="...">
                    {this.formatCount()}
                </span>
                <button className="btn btn-secondary btn-sm">
                    Increment
                </button>
            </React.Fragment>
        );
    }
    ...
```

Rendering Classes Dynamically

```
...
    render() {
        let classes = "badge m-2 badge-";
        classes += this.state.count === 0 ? "warning" : "primary";
        return (
            <React.Fragment>
                <span className={classes}>
                    {this.formatCount()}
                </span>
                <button className="btn btn-secondary btn-sm">
                    Increment
                </button>
            </React.Fragment>
        );
    }
    ...
```

Rendering Classes Dynamically - Refactoring

```
...
    render() {
        return (
            <React.Fragment>
                <span className={this.getBadgeClasses()}>
                    {this.formatCount()}
                </span>
                ...
            </React.Fragment>
        );
    }
    getBadgeClasses() {
        let classes = "badge m-2 badge-";
        classes += this.state.count === 0 ? "warning" : "primary";
        return classes;
    }
```

Rendering Lists

```
...
state = { count: 0, tags: [‘tag1’, ‘tag2’, ‘tag3’] };
render() {
    return (
        <React.Fragment>
            ...
            <ul>
                {
                    this.state.tags.map(tag => <li>{ tag }</li>)
                }
            </ul>
            ...
        </React.Fragment>
    );
}
...
```

Rendering Lists - Add a unique key to each li item

```
...
state = { count: 0, tags: [‘tag1’, ‘tag2’, ‘tag3’] };
render() {
    return (
        <React.Fragment>
            ...
            <ul>
                {
                    this.state.tags.map(tag => <li key={tag}>{ tag
                    }</li>)
                }
            </ul>
            ...
        </React.Fragment>
    );
}
```

Conditional Rendering

```
    ...
    state = { count: 0, tags: ['tag1', 'tag2', 'tag3'] };
    renderTags() {
        if (this.state.tags.length === 0) return <p>There's no tag</p>;
        return <ul>{this.state.tags.map(tag=><li
key={tag}>{tag}</li>)}</ul>;
    }
    render() {
        return (
            <React.Fragment>
                ...
                {this.renderTags()}
            </React.Fragment>
        );
    }
    ...
```

Conditional Rendering - Use logical “AND” operator

```
    ...
    state = { count: 0, tags: [‘tag1’, ‘tag2’, ‘tag3’] };
    renderTags() {
        if (this.state.tags.length === 0) return <p>There’s no tag</p>;
        return <ul>{this.state.tags.map(tag=><li
key={tag}>{tag}</li>)}</ul>;
    }
    render() {
        return (
            <React.Fragment>
                ...
                {this.state.tags.length === 0 && “Please create a new tag”}
                {this.renderTags()}
            </React.Fragment>
        );
    }
    ...
```

Handling Events

```
    ...
    state = { count: 0, tags: [‘tag1’, ‘tag2’, ‘tag3’] };
    ...
    handleIncrement() {
        console.log(‘Increment clicked.’);
    }
    render() {
        return (
            <React.Fragment>
                ...
                <button onClick={this.handleIncrement} className=”btn
btn-secondary btn-sm”>Increment</button>
                ...
            </React.Fragment>
        );
    }
    ...
```

Binding Event Handlers - Problem 1

```
    ...
    state = { count: 0, tags: ['tag1', 'tag2', 'tag3'] };
    ...
    handleIncrement() {
        console.log('Increment clicked:', this); // this is undefined
    }
    render() {
        return (
            <React.Fragment>
                ...
                <button onClick={this.handleIncrement} className="btn
btn-secondary btn-sm">Increment</button>
                ...
            </React.Fragment>
        );
    }
    ...
```

Binding Event Handlers - Problem 2

```
    ...
    constructor() {
        console.log(‘constructor:’, this); // cannot recognize this
    }
    handleIncrement() {
        console.log(‘Increment clicked:’, this); // this is undefined
    }
    render() {
        return (
            <React.Fragment>
                ...
                <button onClick={this.handleIncrement} className=”btn
btn-secondary btn-sm”>Increment</button>
                ...
            </React.Fragment>
        );
    }
    ...
```

Binding Event Handlers - Solution

```
    ...
    constructor() {
        super(); // needed in order to recognize "this" scope
        console.log('constructor:', this);
        this.handleIncrement = this.handleIncrement.bind(this);
    }
    handleIncrement() {
        console.log('Increment clicked:', this); // this is undefined
    }
    render() {
        return (
                ...
                <button onClick={this.handleIncrement} className="btn
btn-secondary btn-sm">Increment</button>
                ...
        );
    }
```

Binding Event Handlers - Another Solution

```
    ...
    handleIncrement = () => {
        console.log('Increment clicked:', this);
    }
    render() {
        return (
                ...
                <button onClick={this.handleIncrement} className="btn
btn-secondary btn-sm">Increment</button>
                ...
        );
    }
```

Updating the State

```
    ...
    handleIncrement = () => {
        //this.state.count++; // this won't work since React won't be aware
        this.setState({ count: this.state.count + 1 });
    }
    render() {
        return (
                ...
                <button onClick={this.handleIncrement} className="btn
btn-secondary btn-sm">Increment</button>
                ...
        );
    }
```

What happens when State changes ?

- Whenever `setState` is called, React will schedule a call to the `render` method.
- We never know when the `render` method will be called by the scheduler (it can be happened within 1s, 5s, etc).
- React will always return a new React element whenever `render` method is called.
- The new React element (Virtual DOM) will be compared with the old Virtual DOM, and new Virtual DOM will replace the old one (only the old tree that has changed, the remaining will be stay the same).

Passing Event Arguments

```
    ...
    handleIncrement = product => {
        console.log(product);
        this.setState({ count: this.state.count + 1 });
    }
    doHandleIncrement = () => {
        this.handleIncrement({ id: 1});
    }
    render() {
        return (
                ...
                <button onClick={this.doHandleIncrement} className="btn
btn-secondary btn-sm">Increment</button>
                ...
        );
    }
    ...
```

Passing Event Arguments - A better way

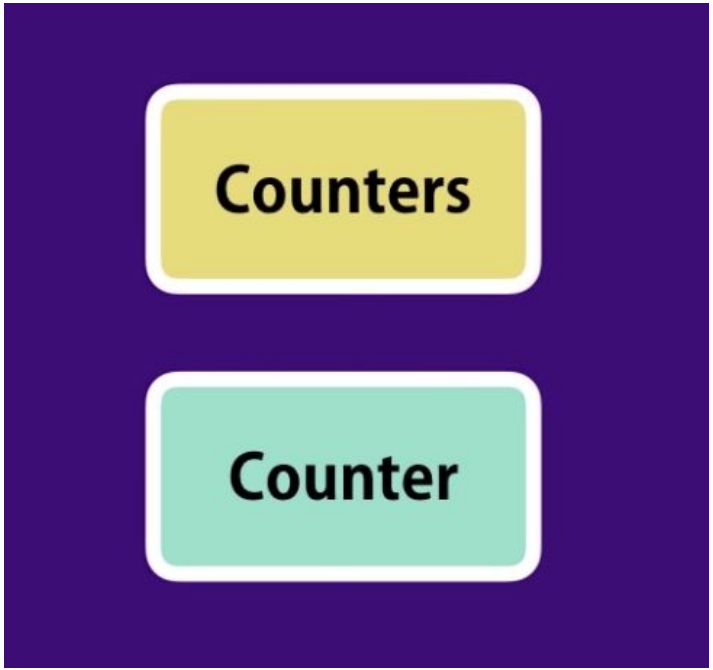
```
    ...
    handleIncrement = product => {
        console.log(product);
        this.setState({ count: this.state.count + 1 });
    }
    render() {
        return (
                ...
                <button onClick={() => this.handleIncrement({id: 1})}
className="btn btn-secondary btn-sm">Increment</button>
                ...
        );
    }
    ...
```

Composing Components

- How to pass data between components ?
- How to raise and handle events ?
- How to have multiple components in sync ?
- Functional components
- Lifecycle Hooks

Composing Components

We’re going to create these following components :




```
<Counters>
        <Counter />
        <Counter />
        …
</Counters>
```

Composing Components

1. Create a new file called **counters.jsx** within the **components** directory
2. Write the following lines in the **counters.jsx**:

```
import React, { Component } from “react”;
import Counter from “./counter”;
export default class Counters extends Component {
    render() {
        return (
            <div>
                <Counter/>
                <Counter/>
                <Counter/>
            </div>
        );
    }
}
```

Composing Components - Refactoring

Instead of hardcoded the `Counter` component, do refactor the **counters.jsx** as follow:

```
import React, { Component } from "react";
import Counter from "./counter";
export default class Counters extends Component {
    state = {
        counters: [{ id: 1, value: 0 }, { id: 2, value: 0 }]
    };
    render() {
        return (
            <div>
                {this.state.counters.map(counter => <Counter key={counter.id}
value={counter.value} selected={true} />)}
            </div>
        );
    }
}
```

Passing Data to Components

What you need to know first :

- Every react components have a property called `props` which is basically a plain JavaScript object which includes all the attributes that is set in the caller component (`Counters` in our previous example)
- So.., refers to our previous example, the 2 attribute values those were set in the **counters.jsx** when calling the `Counter` component, i.e. `value` and `selected` will be the properties of the `props` object. (`key` attribute is not included since it is a special attribute)

Passing Data to Components

Do the following changes to **counter.jsx**:

```
import React, { Component } from “react”;
export default class Counter extends Component {
    state = {
        value: this.props.value
    };
    handleIncrement = product => {
        this.setState({ value: this.state.value + 1 });
    };
    render() {
        ...
    }
    ...
}
```

Passing Children

Inspect the following code at **counters.jsx**:

```
export default class Counters extends Component {
    ...
    render() {
        return (
            <div>
                {this.state.counters.map(counter =>
                    <Counter key={counter.id} value={counter.value}>
                        <h4>Title</h4> // this is the children
                    </Counter>
                )}
            </div>
        );
    }
    ...
}
```

Passing Children - Continued

Call the **children** `props` from **counter.jsx**:

```
import React, { Component } from “react”;
export default class Counter extends Component {
    state = {
        value: this.props.value
    };
    handleIncrement = product => {
        ...
    };
    render() {
        return (<div>
            {this.props.children}
            <span className={this.getBadgeClasses()}>{this.formatValue()}</span>
            ...
        </div>);
    }
    ...
}
```

Passing Children - Continued

Refactor children to be generated dynamically. Change **counters.jsx** as follow:

```
export default class Counters extends Component {
    ...
    render() {
        return (
            <div>
                {this.state.counters.map(counter =>
                    <Counter key={counter.id}>
                        <h4>Counter #{counter.id}</h4>
                    </Counter>
                )}
            </div>
        );
    }
    ...
}
```

Debugging React Apps

- Install Chrome Extension Tool named “React Developer Tools”
- Quick overview on the “React Developer Tools” usage
- Quick overview on the Google Chrome Developer Tools

Props vs State

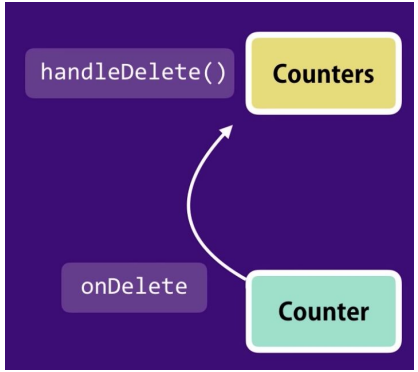
- Props include data that we give into a component. Props is read-only (we cannot change the value once its set).
- State includes data that is local or private to a component, so other components cannot access that particular state (means to say that state completely internal to the component)
- Sometimes, a component might not have state since it can get all the needed data from the props.

Raise an Event

One rule of thumb: "*The component that owns a piece of the state, should be the one that modifying it*".

Demo: we're going to add an ability to delete Counter component from Counters.

1. Add a `delete` button after the `increment` button in the **counter.jsx**.
2. Implement `handleDelete` method on the `Counters` component and raise the "`onDelete`" event from the `Counter` component to `Counters` component (since the `Counters` component was the one that hold the state which contains list of `Counter` component). **Remember the rule of thumb !!!**



Raise an Event - Continued

counters.jsx

```
export default class Counters extends Component {
    state = {counters: [{id: 1, value: 0}, {id: 2, value: 0}]};
    handleDelete = () => {
        console.log('delete button clicked…');
    };
    render() {
        return (<div>
            ...
            <Counter key={counter.id} value={counter.value}
onDelete={this.handleDelete}>
                <h4>Counter #{counter.id}</h4>
            </Counter>
        </div>);
    }
}
```

Raise an Event - Continued

counter.jsx

```
export default class Counter extends Component {
    ...
    render() {
        return (<div>
            {this.props.children}
            ...
            <button onClick={this.props.onDelete}
                className="btn btn-danger btn-sm m-2">Delete</button>
        </div>);
    }
    ...

}
```

Updating the State

counters.jsx

```
export default class Counters extends Component {
    state = {counters: [{id: 1, value: 0}, {id: 2, value: 0}]};
    handleDelete = counterId => {
        console.log('delete button clicked for counter ID', counterId);
    };
    render() {
        return (<div>
            ...
            <Counter key={counter.id} value={counter.value}
                id={counter.id} onDelete={this.handleDelete}>
                <h4>Counter #{counter.id}</h4>
            </Counter>
        </div>);
    }
}
```

Updating the State - Continued

counter.jsx

```
export default class Counter extends Component {
    ...
    render() {
        return (<div>
            {this.props.children}
            ...
            <button onClick={() => this.props.onDelete(this.props.id)}
                className="btn btn-danger btn-sm m-2">Delete</button>
        </div>);
    }
    ...

}
```

Updating the State - Continued

counters.jsx

```
export default class Counters extends Component {
    state = {counters: [{id: 1, value: 0}, {id: 2, value: 0}]};
    handleDelete = counterId => {
        const counters = this.state.counters.filter(c => c.id !== counterId);
        this.setState({ counters }); // ES6 shorthand of this.setState({counters:counters})
    };
    render() {
        return (<div>
            ...
            <Counter key={counter.id} value={counter.value}
                id={counter.id} onDelete={this.handleDelete}>
                <h4>Counter #{counter.id}</h4>
            </Counter>
        </div>);
    }
}
```

Refactoring - Encapsulate related value in an Object

counters.jsx

```
export default class Counters extends Component {
    state = {counters: [{id: 1, value: 0}, {id: 2, value: 0}]};
    handleDelete = counterId => {
        const counters = this.state.counters.filter(c => c.id !== counterId);
        this.setState({ counters }); // ES6 shorthand of this.setState({counters:counters})
    };
    render() {
        return (<div>
            ...
            <Counter key={counter.id} counter={counter}
                onDelete={this.handleDelete}>
                <h4>Counter #{counter.id}</h4>
            </Counter>
        </div>);
    }
}
```

Refactoring - Encapsulate related value in an Object

Do necessary change in the counter.jsx to adapt with the new counter object in props:

```
export default class Counter extends Component {
    state = {value: this.props.counter.value};
    ...
    render() {
        return (<div>
            ...
            <button onClick={() => this.props.onDelete(this.props.counter.id)}
                className="btn btn-danger btn-sm m-2">Delete</button>
        </div>);
    }
    ...
}
```

Single Source of Truth

Let's say we're going to add a `Reset` button which has the ability to reset all counters value to 0 (zero).

counters.jsx

```
…
class Counters extends Component {
    …
    handleReset = () => {
        const counters = this.state.counters.map(c => {
            c.value = 0;
            return c;
        });
        this.setState({ counters });
    };
    render() {
        return (<div>
            <button onClick={this.handleReset} className="btn btn-primary btn-sm m-2">Reset</button>
        </div>);
    }
```

Single Source of Truth - Continued

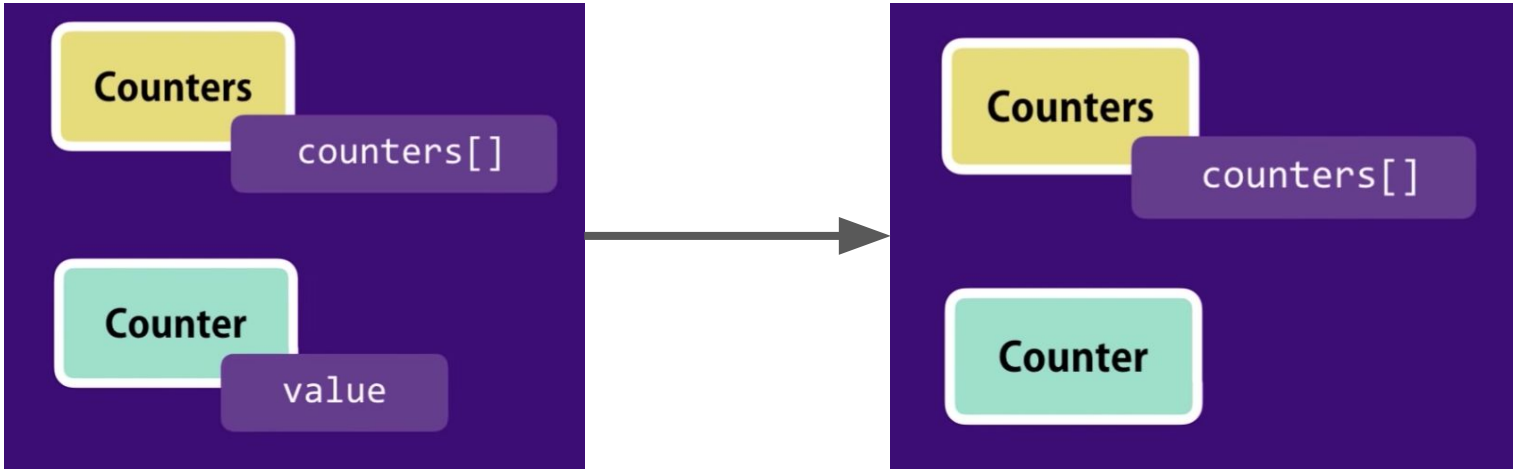
1. Run the app
2. Explain what happened
    a. There is no single source of truth
    b. Data in each component is bound to local state (i.e. not connected to each other)

Tips : Inspect the React Developer Tools

Single Source of Truth - Continued

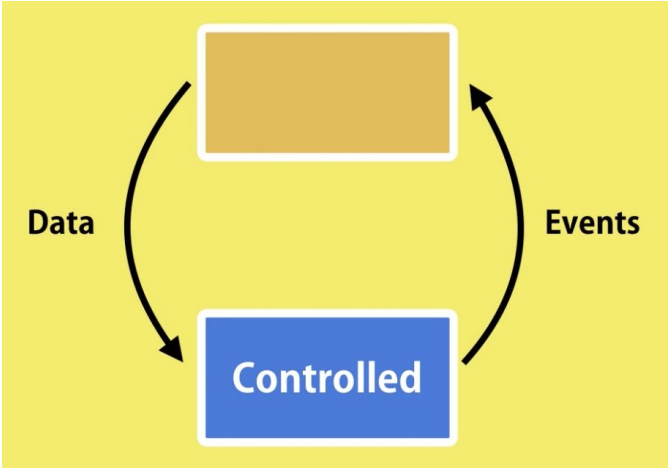
Solution:

Removes the local state on `Counter` component and has a **Single Source of Truth** by just relying on the `props` to receive all the data it's need. In this case, the `Counter` component will become to something known as "**Controlled Component**".



Removing the Local State

1. **Controlled Component** : is a component that doesn't have its own local state. It receives all its data via props, and raise an event whenever data need to be changed. Therefore, this component is entirely controlled by its parent.



Removing the Local State - Continued

counter.jsx

```
...
export default class Counter extends Component {
      render() {
            return (<div>
                  ...
                  <button onClick={() => this.props.onIncrement(this.props.counter)} className="btn
btn-secondary btn-sm m-2">Increment</button>
                  ...
            </div>);
      }
      getBadgeClasses() {
            let classes = "badge m-2 badge-";
            classes += this.props.counter.value === 0 ? "warning" : "primary";
      }
      formatValue() {
            let { value } = this.props.counter;
            ...
      }
}
```

Removing the Local State - Continued

counters.jsx

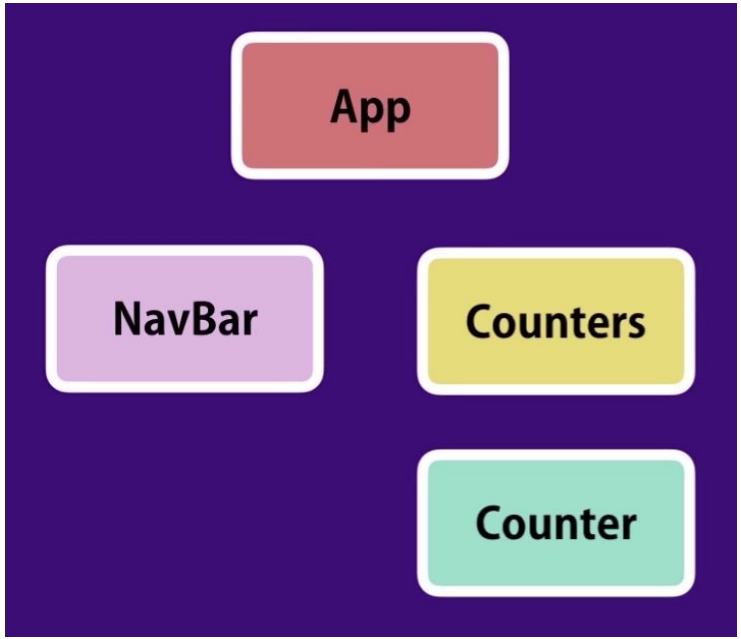
```
...
    handleIncrement = counter => {
        const counters = [...this.state.counters];
        const index = counters.indexOf(counter);
        counters[index] = { ...counter };
        counters[index].value++;
        this.setState({ counters });
    };

    render() {
        return (<div>
            ...
            <Counter key={counter.id} counter={counter} onDelete={this.handleDelete}
                onIncrement={this.handleIncrement}>
                <h4>Counter #{counter.id}</h4>
            </Counter>
        </div>);
    }
...
```

Multiple Components in Sync

We’re going to display the total number of incremented components on the `NavBar`.

First thing first, change the component’s structure to become as depicted in the diagram below:



Multiple Components in Sync - Continued

index.js

```
import React from “react”;
import ReactDOM from “react-dom”;
import “./index.css”;
import App from “./App”;
import registerServiceWorker from “./registerServiceWorker”;
import “bootstrap/dist/css/bootstrap.css”;

ReactDOM.render(<App />, document.getElementById(“root”));
registerServiceWorker();
```

Multiple Components in Sync - Continued

Create a new file named **navbar.jsx** under **components** directory. This is going to be our `NavBar` component. (**Tips:** see https://getbootstrap.com documentation)

navbar.jsx

```
import React, { Component } from “react”;
export default class NavBar extends Component {
    render() {
        return (
            <nav className=”navbar navbar-light bg-light”>
                <a className=”navbar-brand” href=”#”>Navbar</a>
            </nav>
        );
    }
}
```

Multiple Components in Sync - Continued

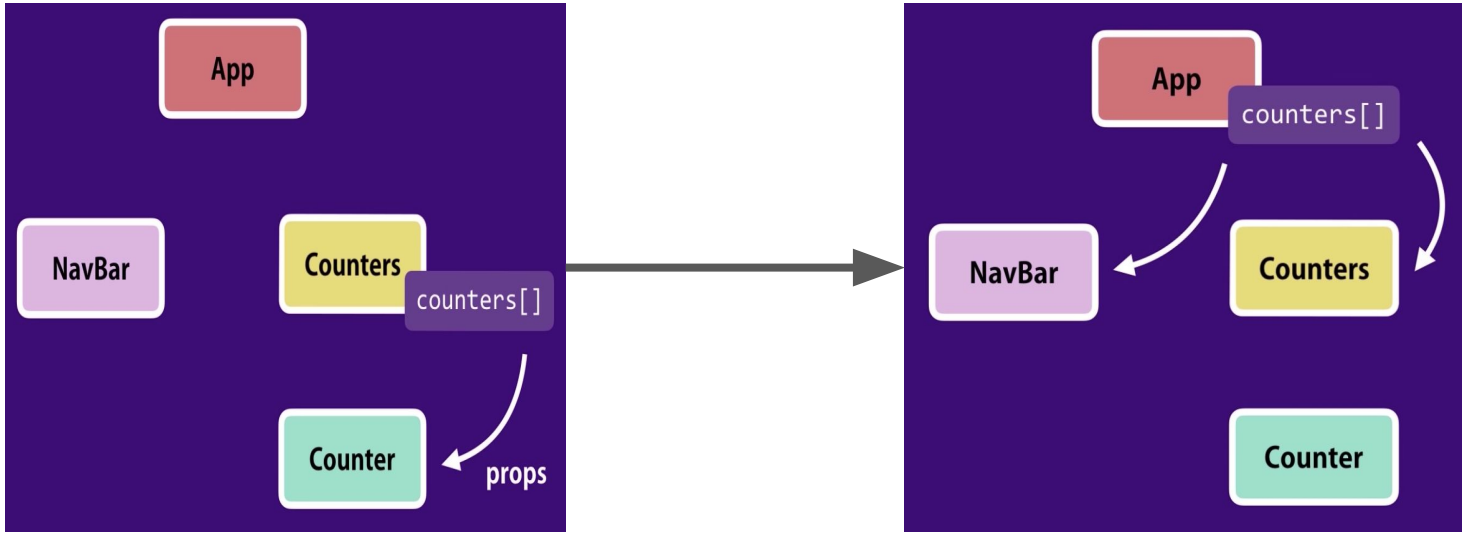
App.js

```
...
import NavBar from './components/navbar';
import Counters from './components/counters';

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

Multiple Components in Sync - Continued

Lifting up the counters state to the `App` component (remember that we're going to display the total of all incremented counters on the `NavBar`).



Lifting State Up - App.js

```
export default Class App extends Component {
      state = { counters: [{id:1, value:0},{id:2, value:0}] };
      handleIncrement = counter => {
            const counters = [...this.state.counters];
            const index = counters.indexOf(counter);
            counters[index] = { ...counter };
            counters[index].value++;
            this.setState({ counters });
      };
      handleDelete = counterId => {
            const counters = this.state.counters.filter(c => c.id !== counterId);
            this.setState({ counters });
      };
      handleReset = () => {
            const counters = this.state.counters.map(c => {
                  c.value = 0;
                  return c;
            });
            this.setState({ counters });
      };
```

Lifting State Up - App.js (Continued)

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

Lifting State Up - navbar.jsx

```
export default 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">
                        {this.props.totalCounters}
                    </span>
                </a>
            </nav>
        );
    }
}
```

Lifting State Up - counters.jsx

```
export default class Counters extends Component {

      render() {
            return (
                  <div>
                        <button onClick={this.props.onReset}
                              className="btn btn-primary btn-sm m-2">
                              Reset
                        </button>
                        {this.props.counters.map(counter => (
                              <Counter key={counter.id} counter={counter}
                                    onDelete={this.props.onDelete}
                                    onIncrement={this.props.onIncrement}>
                                    <h4>Counter #{counter.id}</h4>
                              </Counter>
                        ))}
                  </div>
            );
      }
}
```

Stateless Functional Components

We can change all of our components which only have `render` method to be a "**Stateless Functional Component**".

E.g. change our `NavBar` component (**navbar.jsx**) to become as shown below:

```
// 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">
                              {props.totalCounters}
                        </span>
                  </a>
            </nav>
      );
};
export default NavBar;
```

Destructuring Arguments

Refactor **navbar.jsx** to use **Destructuring Arguments** (ES6 feature).

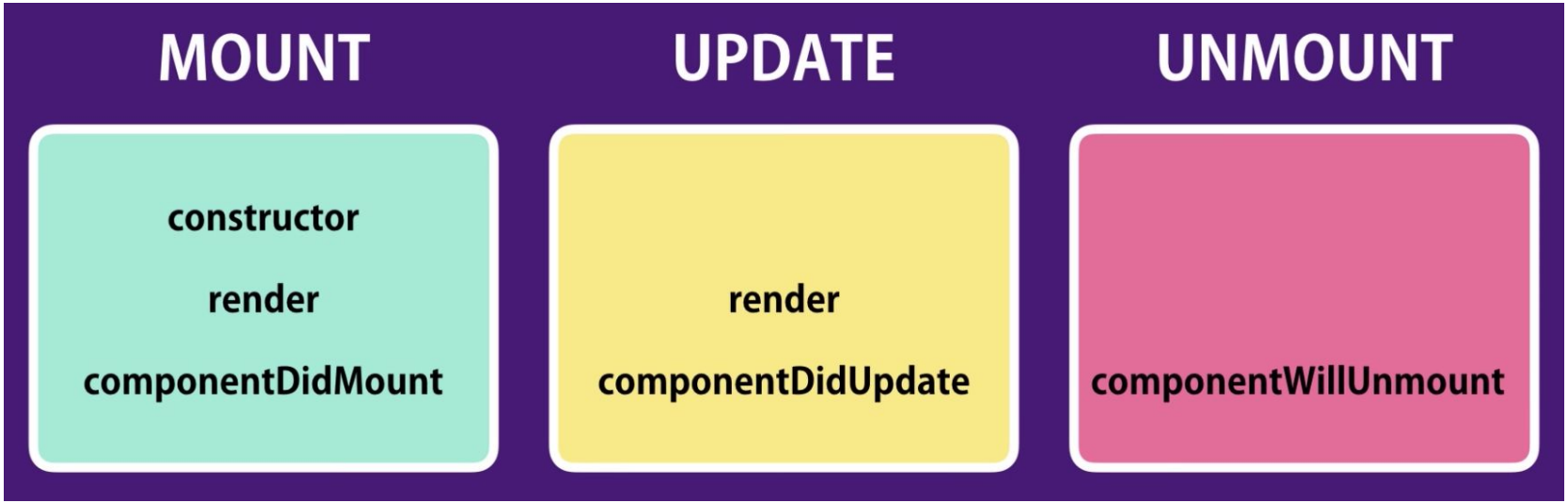
```
const NavBar = ({ totalCounters }) => {
    return (
        <nav className="navbar navbar-light bg-light">
            <a className="navbar-brand" href="">
                Navbar{" "}
                <span className="badge badge-pill badge-secondary">
                    {totalCounters}
       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

Destructuring Arguments - Continued

Refactor **counters.jsx** to use **Destructuring Arguments** (ES6 feature).

```
export default class Counters extends Component {
    render() {
        const { onReset, counters, onDelete, onIncrement } = this.props;
        return (<div>
            <button onClick={onReset} className="btn btn-primary btn-sm m-2">
                Reset
            </button>
            {counters.map(counter => (
                <Counter key={counter.id} counter={counter}
                    onDelete={onDelete} onIncrement={onIncrement}>
                    <h4>Counter #{counter.id}</h4>
                </Counter>
            )}
        </div>);
    }
}
```

Lifecycle Hooks



Q & A / Discussion