Unit 3 - ReactJS

Class notes by Vibha Masti

Feedback/corrections: vibha@pesu.pes.edu

0.0 Table of Contents

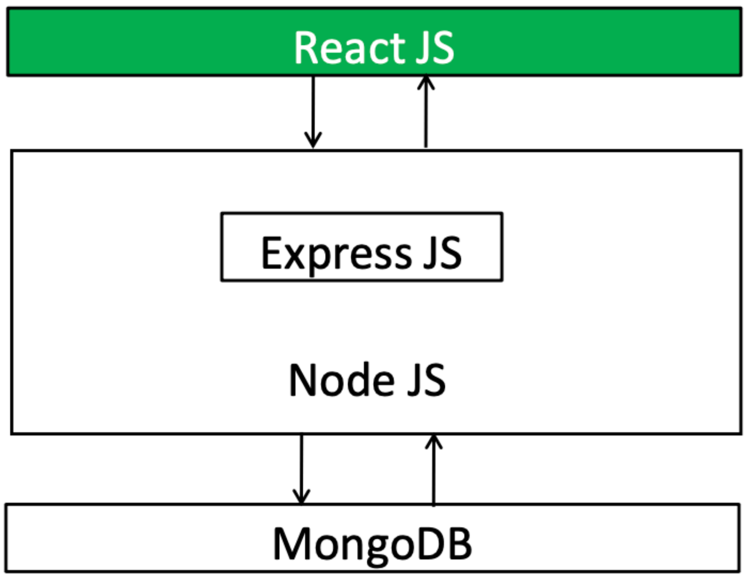
1.0 Introduction to MERN Stack

1. MongoDB
    - Database server
    - Bottom tier of MERN stack
    - Application data store
    - JSON docs can be stored
    - Key-value pairs
    - Non-SQL

2. ExpressJS
    - ExpressJS and NodeJS - server side frameworks
    - Middle of MERN stack
    - Express: URL Routing
    - Build APIs

3. ReactJS
    - This unit
    - Frontend JS Library (mainly SPA)
    - Components
    - Connect to backend server
    - Render to HTML
    - Stateful, data-driven interfaces
    - Forms, error handling, events, lists

4. NodeJS
    - Web server
    - Use NodeJS MongoDB drivers
    - Using callbacks and promises



Why MERN and not MEAN?

- Angular - MVC, heavy, learning curve
- React - easier, library

1.1 Introduction to React

- Released by Facebook and Instagram to help make building SPAs (single page applications) easier
- DOM manipulation is faster (using virtual DOM)
- Uses components for visual elements

Babel and JSX

- Babel is a JSX to JavaScript converter that allows us to create React elements in XML/HTML syntax
- JSX gets converted to `React.createElement()`

JSX

```
ReactDOM.render(
    <div>
        <h1>Captain America</h1>
        <h1>Iron Man</h1>
        <h1>Thor</h1>
    </div>,
    destination
);
```

JavaScript

```
ReactDOM.render(
    React.createElement("div", null,
        React.createElement( "h1", null, "Captain America" ),
        React.createElement ( "h1", null, "Iron Man" ),
        React.createElement ( "h1", null, "Thor" ),
    ),
    destination
);
```

1.2 Getting Started with React

First Program

Include the following libraries in the `<script></script>` tags

```
<!-- React JS libraries -->

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

<!-- Babel -->
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
```

1. Tip: if you are using VSCode, type in `html:5` and press enter to get a basic HTML5 skeleton

2. Be sure to include the sources in the header `<script></script>` tags
3. Create an empty `<div></div>` tag to manipulate using `ReactDOM`
4. Add a script tag with text type `text/babel` and start writing your react code!

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Introduction to React</title>

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

</head>
<body>
    <div id="container"></div>
    <script type="text/babel">
        var des = document.querySelector("#container");
        ReactDOM.render(
            {/* JSX code */}
            <h1> React World </h1>,
            des
        );
    </script>
</body>
</html>
```

3. Note: the JSX code can be written as

```
React.createElement("h1", {color: red}, "React World!!"),
```

2.0 Introduction to Components

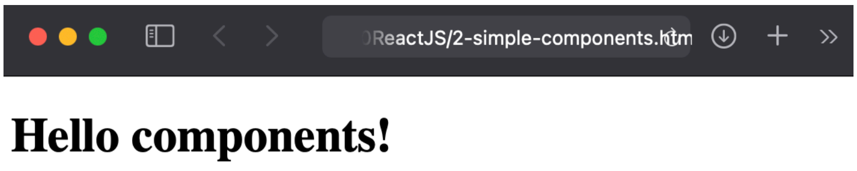
Simple Components

- Components are written as classes that inherit from `React.Component`
- A render method must be written that returns JSX
- Call the `ReactDOM.render()` method with an element of the newly-created component

Inside the body `<script type="text/babel"></script>` tags

```
var des = document.querySelector("#container");

class HelloWorld extends React.Component {
    render() {
        return <h1> Hello components! </h1>
    }
}

ReactDOM.render(
    <HelloWorld />,
    des
);
```

Rendered in a browser



Hello components!

Parameterised Components (using this.props)

- JSX code within `{}` is JS
- `this.props` refers to the properties passed in the JSX from the `ReactDOM.render()` function

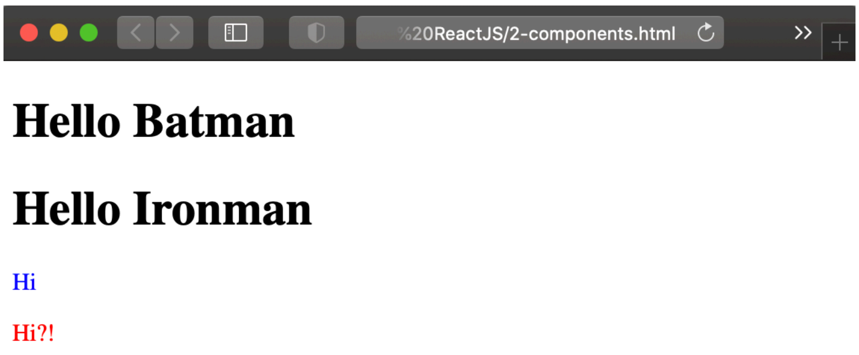
Inside the body `<script type="text/babel"></script>` tags

```
var des = document.querySelector("#container");

class HelloWorld extends React.Component {
    render() {
        {/* This is a JSX comment */}
        {/* {this.props.greet} is displayed in the <h1></h1>
        tags on the browser*/}
        return <h1> Hello {this.props.greet} </h1>
    }
}

class Hello extends React.Component {
    render() {
        return (
            {/* A component can have child elements like in HTML */}
            <div className={this.props.type}>
                {/* this.props.children contains the text within the tags,
                not an array of children */}
                <p>{this.props.children}</p>
            </div>
        );
    }
}

ReactDOM.render(
    <div>
        {/* property greet is passed */}
        <HelloWorld greet="Batman"/>
        <HelloWorld greet="Ironman"/>

        <Hello type="greeting">
            Hi
        </Hello>
        <Hello type="shocking">
            Hi?!
        </Hello>
    </div>,
    des
);
```

Inside `<style></style>` tags in the head of the page, we can add some basic styling

```
<style>
    .greeting {
        color: blue
    }

    .shocking {
        color: red
    }
</style>
```

Rendered in a browser



Hello Batman

Hello Ironman

Hi

Hi?!

2.2 Styling Components

Hard-Coded Approach

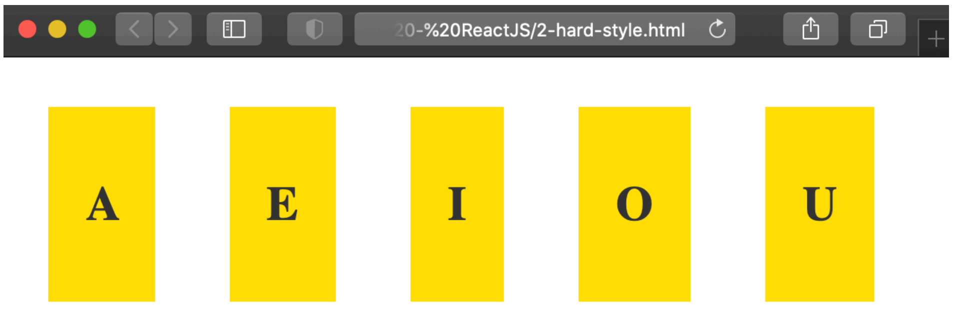
- To style various components, we can add CSS style to the `<style></style>` tags in the head
- This is not the best way to do it

```
.letter {
    background-color: #ffde00;
    color: #333;
    display: inline-block;
    padding: 25px;
    margin: 25px;
}
```

Inside the body `<script type="text/babel"></script>` tags

```
var des = document.querySelector("#container");

class Letter extends React.Component {
    render() {
        return <div className="letter">
            <h1>{this.props.children}</h1>
        </div>;
    }
}

ReactDOM.render(
    <div>
        <Letter> A </Letter>
        <Letter> E </Letter>
        <Letter> I </Letter>
        <Letter> O </Letter>
        <Letter> U </Letter>
    </div>,
    des
);
```

Rendered in a browser



Styling it the React Way

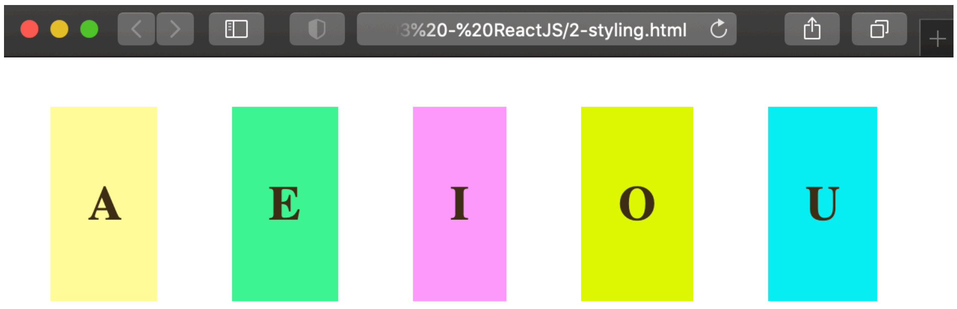
- Create a style object in the `Letter` component class
- Syntax similar to CSS but uses camelCase
- Reads backgroundColor property from `<Letter></Letter>` tags
- No `<style></style>` tags in the head

Inside the body `<script type="text/babel"></script>` tags

```
var des = document.querySelector("#container");
```

```

class Letter extends React.Component {
    render() {
        var letterStyle = {
            backgroundColor: this.props.backgroundColor,
            color: '#432e19',
            display: 'inline-block',
            padding: '25px',
            margin: '25px'
        };
        return <div style={letterStyle}>
            <h1>{this.props.children}</h1>
        </div>;
    }
}

ReactDOM.render(
    <div>
        <Letter backgroundColor='#ffff99'> A </Letter>
        <Letter backgroundColor='#45f594'> E </Letter>
        <Letter backgroundColor='#ff99ff'> I </Letter>
        <Letter backgroundColor='#dbfb10'> O </Letter>
        <Letter backgroundColor='#00f2f4'> U </Letter>
    </div>,
    des
);
```

Rendered in a browser



3.0 Complex Components

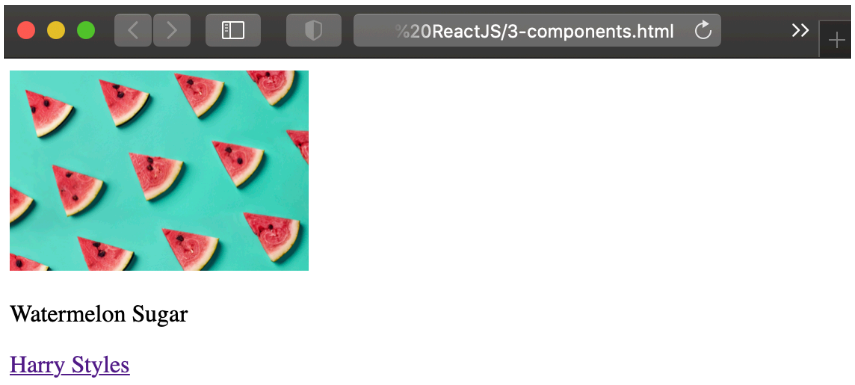
- Good for reusability and composability
- Components that contain components
- For example, a component called `SearchResult` that contains the components `ResultImage`, `ResultCaption` and `ResultLink`

Hard-Coded

Inside the body `<script type="text/babel"></script>` tags

```
class ResultImage extends React.Component {
    render() {
        return (
            <img src="images/watermelon.jpg" width="200px"></img>
        );
    }
}

class ResultCaption extends React.Component {
    render() {
        return (
            <p>Watermelon Sugar</p>
        );
    }
}

class ResultLink extends React.Component {
    render() {
        return (
            <a href="https://www.youtube.com/watch?v=E07s5ZYygMg">Harry Styles</a>
        );
    }
}

class SearchResult extends React.Component {
    render() {
        return (
            <div>
                <ResultImage/>
                <ResultCaption/>
                <ResultLink/>
            </div>
        );
    }
}

ReactDOM.render(
    <SearchResult/>,
    document.querySelector("#container")
);
```

Rendered in a browser



Watermelon Sugar

Harry Styles

Better Composite Components

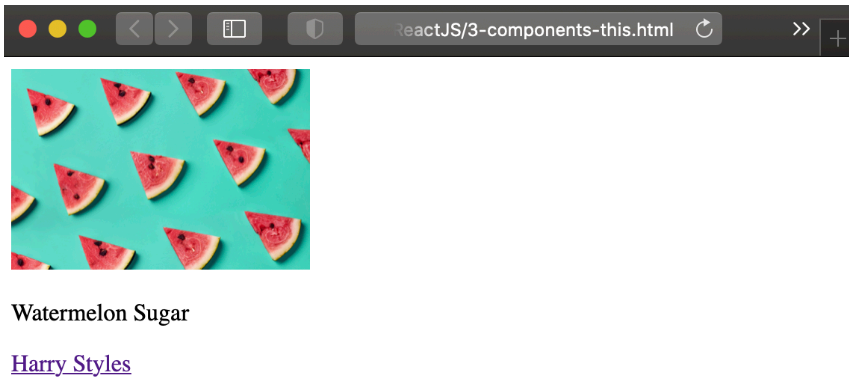
- To remove the hardcoding, the properties like `src`, `href` etc., have to be passed on using `this.props` from the parent Components to its sub components, as follows
- If desired, styling may be added as previously discussed

Inside the body `<script type="text/babel"></script>` tags

```
class ResultImage extends React.Component {
    render() {
        return (
            <img src={this.props.src} width={this.props.width}></img>
        );
    }
}

class ResultCaption extends React.Component {
    render() {
        return (
            <p>{this.props.children}</p>
        );
    }
}

class ResultLink extends React.Component {
    render() {
        return (
            <a href={this.props.href}>Harry Styles</a>
        );
    }
}
```

```

class SearchResult extends React.Component {
    render() {
        return (
            <div>
                {/* Tedious way to transfer properties */}
                <ResultImage src={this.props.src} width={this.props.width}/>
                <ResultCaption>{this.props.children}</ResultCaption>
                <ResultLink href={this.props.href}/>
            </div>
        );
    }
}


ReactDOM.render(
    <SearchResult src="images/watermelon.jpg" width="200px" href="https://www.youtube.com/watch?v=E07s5ZYygMg">Watermelon Sugar</SearchResult>,
    document.querySelector("#container")
);
```

Rendered in a browser



Watermelon Sugar

Harry Styles

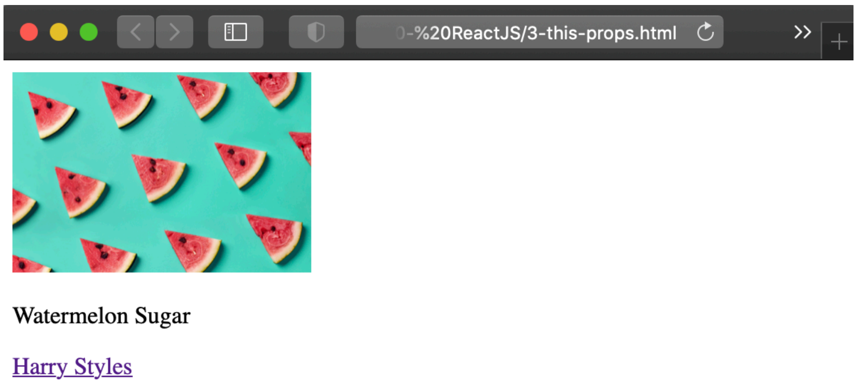
Transferring Properties Better

- To simplify the tedious process of transferring components, we use the spread operator - `...`

Inside the body `<script type="text/babel"></script>` tags

```
class ResultImage extends React.Component {
    render() {
        return (
            <img src={this.props.src} width={this.props.width}></img>
        );
    }
}

class ResultCaption extends React.Component {
    render() {
        return (
            <p>{this.props.caption}</p>
        );
    }
}

class ResultLink extends React.Component {
    render() {
        return (
            <a href={this.props.href}>{this.props.children}</a>
        );
    }
}

class SearchResult extends React.Component {
    render() {
        return (
            <div className="result">
                <ResultImage {...this.props}/>
                <ResultCaption {...this.props}/>
                <ResultLink {...this.props}/>
            </div>
        );
    }
}

ReactDOM.render(
    <SearchResult src="images/watermelon.jpg"
        width="200px" caption="Watermelon Sugar"
        href="https://www.youtube.com/watch?v=E07s5ZYygMg">
        Harry Styles
    </SearchResult>,

    document.querySelector("#container")
);
```

Rendered in a browser



Watermelon Sugar

Harry Styles

4.0 Stateful Components

- Up until now, we have only dealt with static/stateless components that do not undergo any state changes
- Components may need to change based on user actions, timers, responses from servers etc.

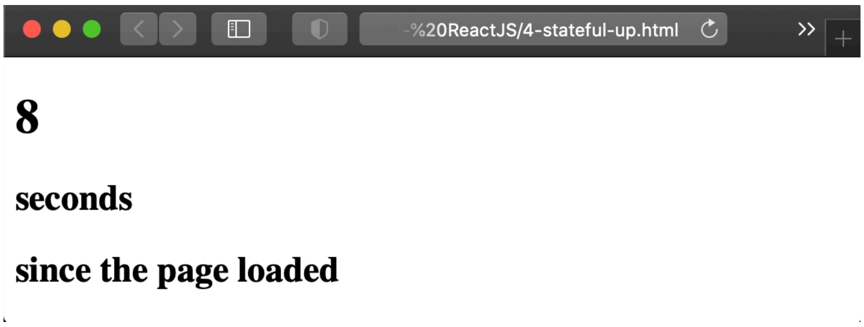
Basic Counter Component

- Shows number of seconds that the user has been on the page for
- We create a `Component` Counter with a `constructor()`, `componentDidUpdate()` method, `componentDidMount()` method, `timer()` method and a `render()` method
- The `constructor(props, context)` is to initialise the counter and `this.state` and should always call the `super(props, context)` constructor
- The method `componentDidMount()` can be used to start the counter and set the timer
- The `timer()` method calls the `setState()` function, which is what makes the component stateful
-

Inside the body `<script type="text/babel"></script>` tags

```
class Counter extends React.Component {
    constructor (props, context) {
        super(props, context);
        this.state = {
            seconds: 0
        };
        this.timer = this.timer.bind(this);
    }

    componentDidUpdate() {
        console.log("Updating!!");
    }

    componentDidMount() {
        this.t = setInterval(this.timer, 1000);
    }

    componentWillUnmount() {
        clearInterval(this.t);
        console.log("Stopped!!!");
    }

    timer () {
        this.setState((prevState) => {
            return {
                seconds: prevState.seconds + 1
            }

        })
    }

    render() {
        return (
            <h1>{this.state.seconds}</h1>
        )
    }
}

class CounterDisplay extends React.Component {
    render() {
        return (
            <div>
                <Counter/>
                <h2> seconds </h2>
                <h2> since the page loaded </h2>
            </div>
        )
    }
}


ReactDOM.render(
    <CounterDisplay/>,
    document.querySelector("#container")
);
```

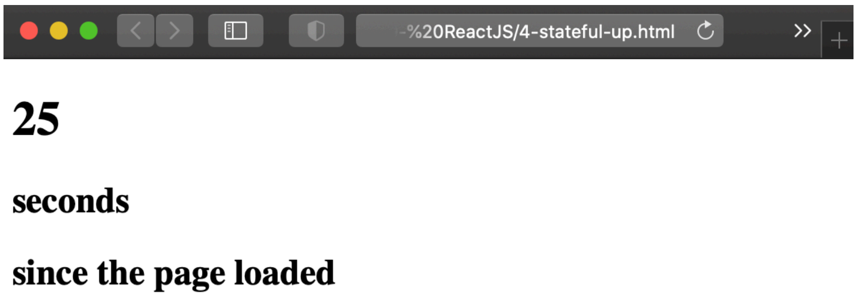
Rendered in a browser



8

seconds

since the page loaded

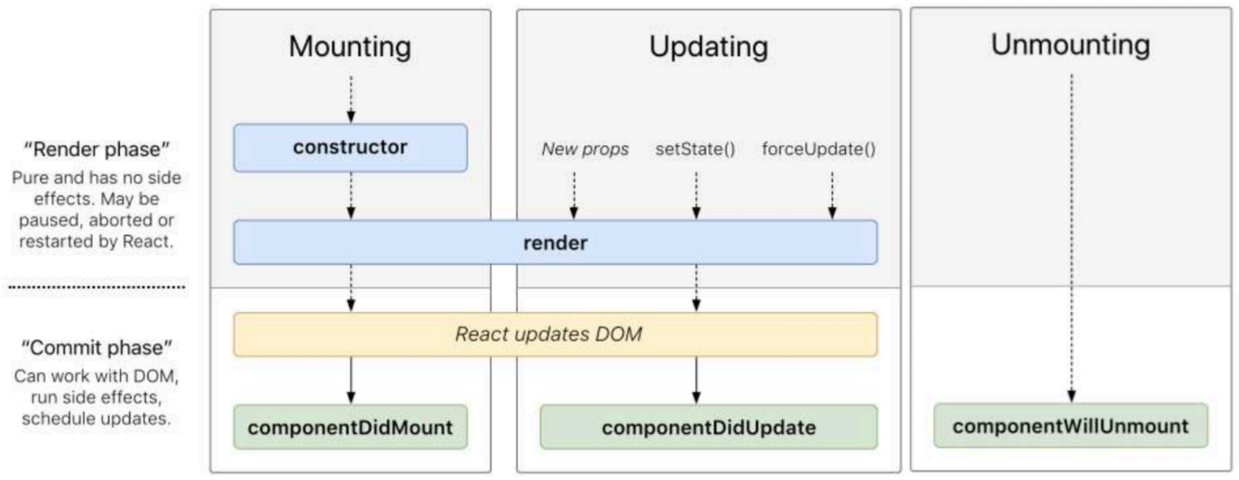


25

seconds

since the page loaded

Component Life Cycle



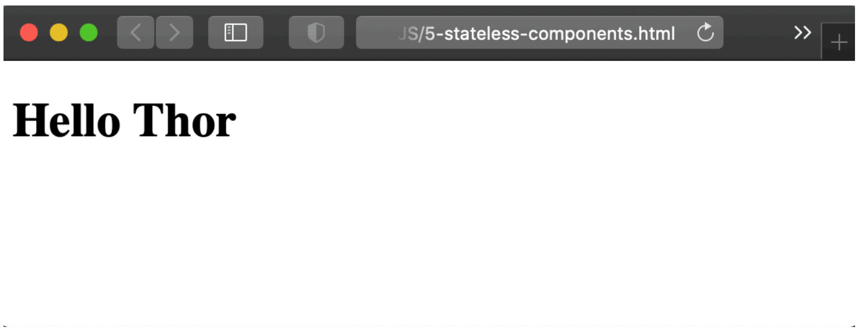
5.0 Stateless Components

- Class-based components are quite heavy
- For some stateless components, function-based components can be used
- Properties can be passed as parameters, not using `this`

Inside the body `<script type="text/babel"></script>` tags

```
function Stuff(props) {
    return (
        <div>
            <h1>Hello {props.name}</h1>
        </div>
    );
}

ReactDOM.render(
    <Stuff name="Thor"/>,
    document.querySelector('#container')
);
```

Rendered in a browser



Hello Thor

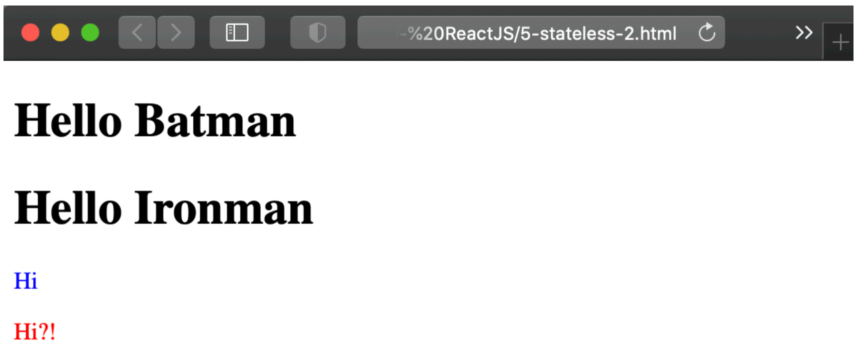
More on Stateless Components

Inside the body `<script type="text/babel"></script>` tags

```
var des = document.querySelector("#container");

function HelloWorld(props) {
    return <h1> Hello {props.greet} </h1>
}

function Hello(props) {
    return (
        <div className={props.type}>
            <p>{props.children}</p>
        </div>
    );
}

```

```

ReactDOM.render(
    <div>
        <HelloWorld greet="Batman"/>
        <HelloWorld greet="Ironman"/>
        <Hello type="greeting">
            Hi
        </Hello>
        <Hello type="shocking">
            Hi?!
        </Hello>
    </div>,
    des
);
```

Rendered in a browser



Hello Batman

Hello Ironman

Hi

Hi?!

6.0 Key Property

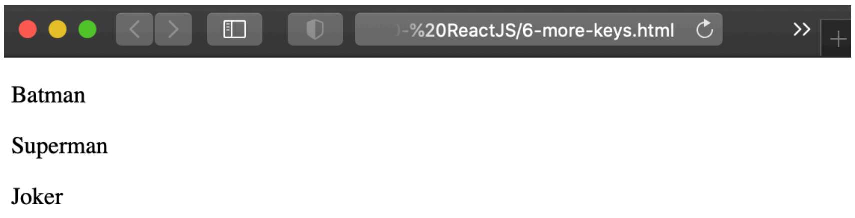
- When returning an array or list of elements, the individual elements should be uniquely identified by a key property
- Helps React identify each element in the list
- Unique key properties for each child in an array or iterator

Hard-Coded Method

Inside the body `<script type="text/babel"></script>` tags

```
function Stuff() {
    return (
        [
            <p key="1">Batman</p>,
            <p key="2">Superman</p>,
            <p key="3">Joker</p>,
        ]
    )
}
```

Rendered in a browser



Batman

Superman

Joker

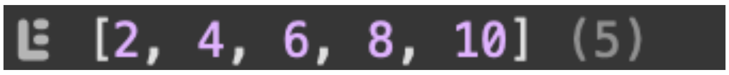
Using Map Method

- Map method on an array or list calls a callback function for each element of the array

Example of `map()`

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
```

Output



```
[2, 4, 6, 8, 10] (5)
```

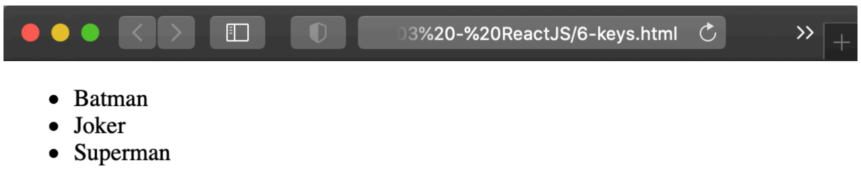
Using key property

- Return a `<li>` element for each item
- If listItems does not have a `key` property, error

Inside the body `<script type="text/babel"></script>` tags

```
function NameList(props) {
    const names = props.names;
    const listItems = names.map((name, index) => <li key={index}>{name}
</li>);

    return (
        <ul>{listItems}</ul>
    )
}

const names = ["Batman", "Joker", "Superman"];

ReactDOM.render(
    <NameList names={names}/>,
    document.querySelector('#container')
);
```

Rendered in a browser



- Batman
- Joker
- Superman

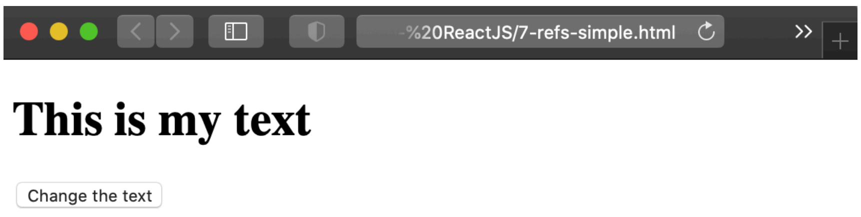
7.0 References

- Instead of accessing DOM elements in JSX, can use `refs`
- Can be used to induce changes in Components or Elements after they are rendered
- `ref`s are callback functions passed as properties

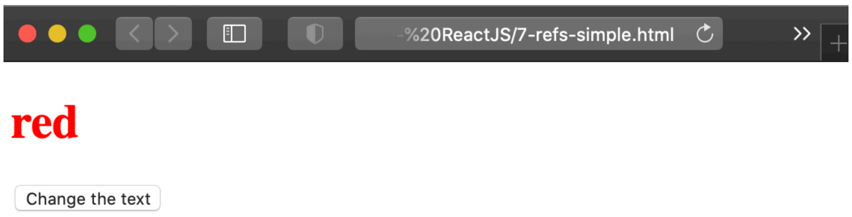
Inside the body `<script type="text/babel"></script>` tags

```
var colors = ['yellow', 'red', 'green', 'blue', 'orange']
class CustomText extends React.Component {
    constructor(props, context) {
        super(props, context)
        this.myText = null;
        this.setTextRef = element => {
            /* this.myText is a reference to the element
            and can be used to perform raw DOM operations */
            this.myText = element;
        }
        this.changeText = event => {
            this.myText.innerHTML = "Changed";
            var i = Math.floor(Math.random()*5);
            this.myText.innerHTML = colors[i];
            this.myText.style.color = colors[i];
        }
    }


    render() {
        return (
            <div>
                <h1 ref={this.setTextRef}>This is my text</h1>
                <input type="button" value="Change the text" onClick={this.changeText}/>

            </div>
        )
    }
}

ReactDOM.render(
    <CustomText/>,
    document.querySelector('#container')
);
```

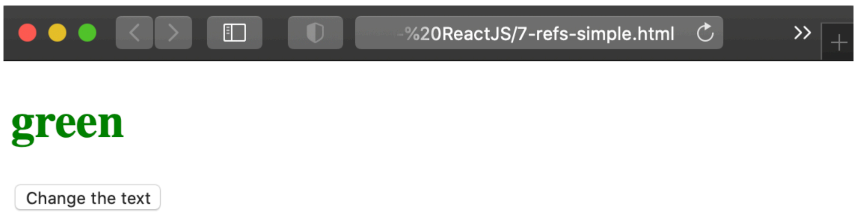
Rendered in a browser



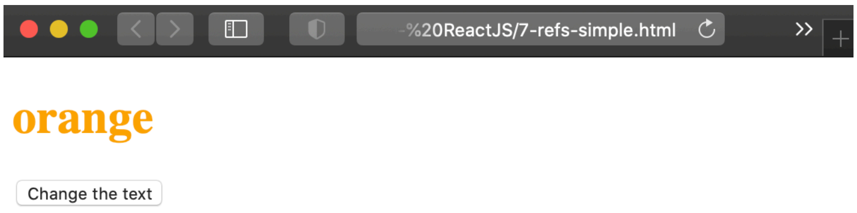
This is my text



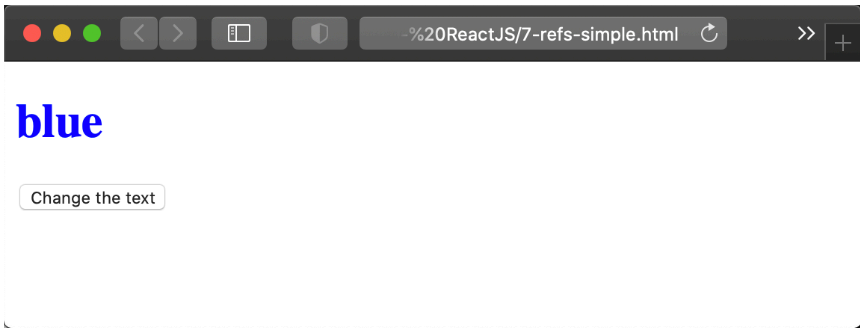
red



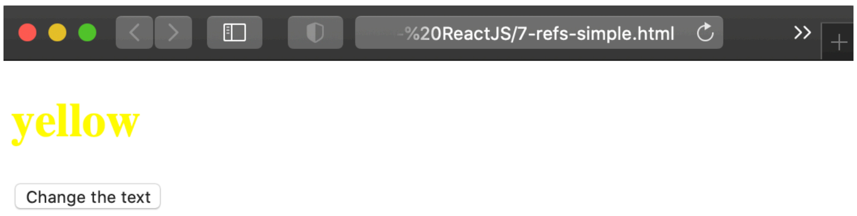
green



orange



blue



yellow

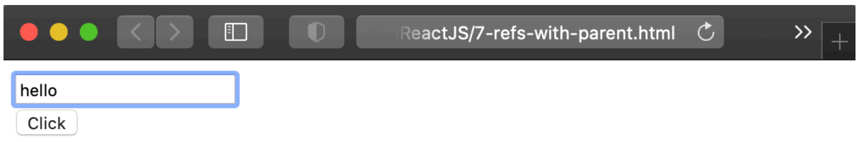
7.1 Passing References

- Parents can pass a `ref` callback to its child element to get a reference to the child element

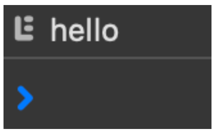
Inside the body `<script type="text/babel"></script>` tags

```
function CustomInput (props) {
    return (
        <div>
            <input ref={props.inputRef} type="text"/>
        </div>
    )
}

class Parent extends React.Component {
    constructor(props, context) {
        super(props, context);

        this.doClick = event => {
            this.inputText.focus();
            console.log(this.inputText.value);
        }
    }

    render() {
        return (
            <div>
                <CustomInput inputRef={el => {this.inputText = el}}/>
                <button onClick={this.doClick}>Click</button>
            </div>
        )
    }
}

ReactDOM.render(
    <Parent/>,
    document.querySelector('#container')
);
```

Rendered in a browser



Console



hello

8.0 Events

- Similar to DOM events, but with some syntax differences
- React events use camelCase and not lowercase (as in DOM events)
- With JSX, event handler is a function and not a string
- Event objects are of type `SyntheticEvent` object

`SyntheticEvent` Object Characteristics

- `SyntheticEvent` is a wrapper around the `DOMEvent` object
- Event handlers are registered at the time of rendering, rather than using `addEventListener` after the element has been created
- Returning false does not prevent the default browser behaviour
- `e.preventDefault()` or `e.stopPropagation()` should be used

`SyntheticEvent` Object properties

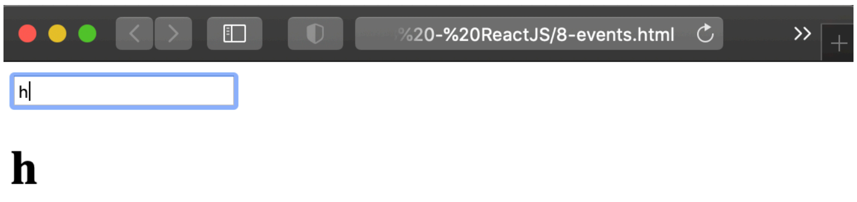
- `booleanbubbles`
- `booleancancelable`
- `DOMEventTargetcurrentTarget`
- `booleandefaultPrevented`
- `numbereventPhase`

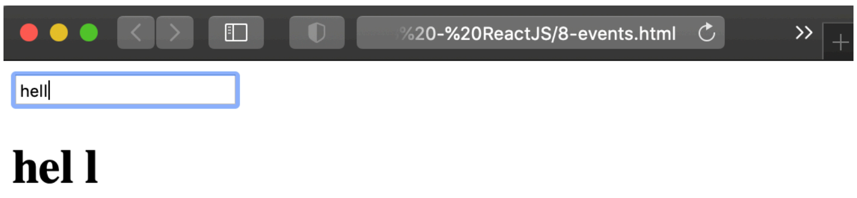
- `booleanisTrusted`
- `DOMEventnativeEvent`
- `voidpreventDefault()`
- `booleanisDefaultPrevented()`
- `voidstopPropagation()`
- `booleanisPropagationStopped()`
- `voidpersist()`
- `DOMEventTargettarget`
- `numbertimeStamp`
- `stringtype`

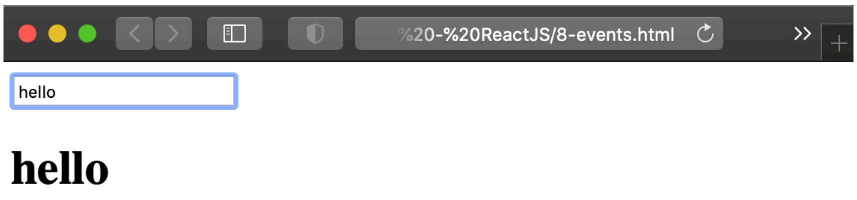
Inside the body `<script type="text/babel"></script>` tags

```
class MyDiv extends React.Component {
    constructor(props) {
        super(props);
        this.myText = null;
        this.showChar = event => {
            var txt;
            this.myText.innerHTML = event.target.value + ' ';

            if (event.shiftKey) {
                txt = '<span style="color:red">'+event.key+'</span>';
            }
            else {
                txt = event.key;
            }
            this.myText.innerHTML += txt;
        }
        this.setTextRef = element => {
            this.myText = element;
        }
    }
    render() {
        return (
            <div>

                <input onKeyPress={this.showChar} type='text'/>
                <h1 ref={this.setTextRef}/>
            </div>
        );
    }
}

ReactDOM.render(
    <MyDiv/>,
    document.querySelector('#container')
);
```

Rendered in a browser







9.0 Forms

- Two main functionalities: when input value is change (`onChange` event) and when the form is submitted (`onSubmit` event)
- Form Data in React is usually handled by Components by storing them in `state` object, such Form components are called **Controlled Components**

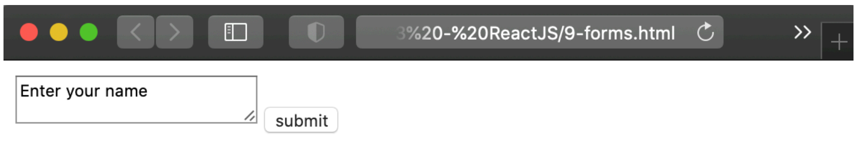
Controlled Components

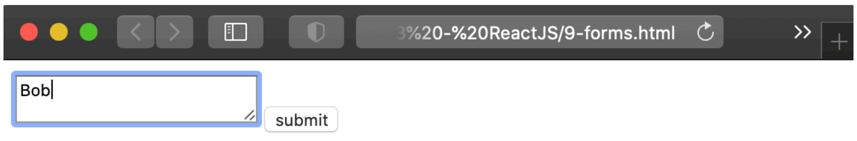
- The value property of the three types of form elements `<input>`, `<textarea>` and `<select>` are controlled by React using the `state` and updated only using `setState`
- The value is updated in the state when `onChange` event is triggered on the form element (which calls `setState`)
- The value is also set to the state property to keep it updated at all times (single source of truth)

Inside the body `<script type="text/babel"></script>` tags

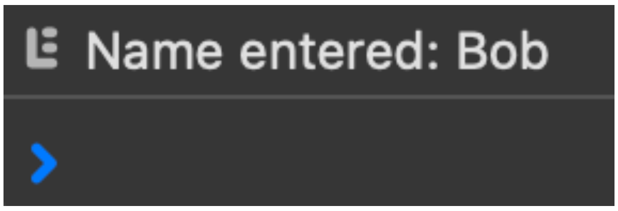
```
var txt, rv;

class ReadName extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            value: 'Enter your name'
        };

        /* Instead of defining
        functions in constructor */
        this.handleChange = this.handleChange.bind(this);
        this.handleSubmit = this.handleSubmit.bind(this);
    }

    handleChange = event => {
        this.setState({
            value: event.target.value
        });
    }

    handleSubmit = function (event) {
        console.log("Name entered: " + this.state.value);
        event.preventDefault();
    }

    render() {
        return (
            <form onSubmit={this.handleSubmit}>
                <textarea value={this.state.value} onChange=
{this.handleChange}></textarea>
                <input type="submit" value="submit"/>
            </form>
        )
    }
}

ReactDOM.render(
    <ReadName/>,
    document.querySelector('#container')
);
```

Rendered in a browser





Console



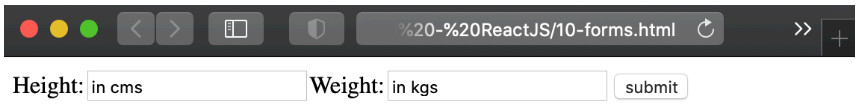
```
Name entered: Bob
```

More on Forms

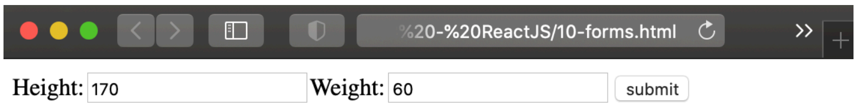
Inside the body `<script type="text/babel"></script>` tags

```
class BMICalc extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            height: 'in cms',
            weight: 'in kgs',
            bmi: '0',
            bmistat: '0'
        };

        /* Instead of defining
        functions in constructor */
        this.handleChangeHeight = this.handleChangeHeight.bind(this);
        this.handleChangeWeight = this.handleChangeWeight.bind(this);
        this.handleSubmit = this.handleSubmit.bind(this);
    }

    handleChangeHeight = event => {
        var height_mtrs = event.target.value/100;
        var bmi = this.state.weight/(height_mtrs*height_mtrs);
        var bmistat = null;
        if (bmi < 19) {
            bmistat = 'Underweight'
        }
        else if (bmi < 26) {
            bmistat = 'Normal'
        }
        else {
            bmistat = 'Overweight'
        }
        this.setState({
            height: event.target.value,
            weight: this.state.weight,
            bmi: bmi,
            bmistat: bmistat
        });
    }

    handleChangeWeight = event => {
        var height_mtrs = this.state.height/100;
        var bmi = event.target.value/(height_mtrs*height_mtrs);
        var bmistat = null;
        if (bmi < 19) {
            bmistat = 'Underweight'
        }
        else if (bmi < 26) {
            bmistat = 'Normal'
        }
        else {
            bmistat = 'Overweight'
        }
        this.setState({
            height: this.state.height,
            weight: event.target.value,
            bmi: bmi,
            bmistat: bmistat
        });
    }

    handleSubmit = function (event) {
        console.log("Height entered: " + this.state.height);
        console.log("Weight entered: " + this.state.weight);
        console.log("BMI: " + this.state.bmi);
        console.log("BMI status: " + this.state.bmistat);
        event.preventDefault();
    }

    render() {
        return (
            <form onSubmit={this.handleSubmit}>
                <label>
                    Height:
                </label>
                <input value={this.state.height}
                    onChange={this.handleChangeHeight} type="text">
</input>

                <label>
                    Weight:</label>
                <input value={this.state.weight}
                    onChange={this.handleChangeWeight} type="text">
</input>
                <input type="submit" value="submit"/>
            </form>
        )
    }
}

ReactDOM.render(
    <BMICalc/>,
    document.querySelector('#container')
);
```

Rendered in a browser

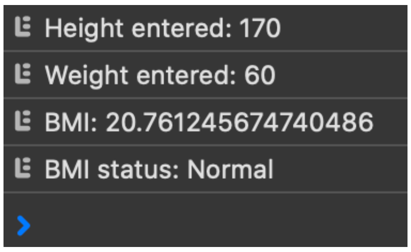


Height: in cms Weight: in kgs



Height: 170 Weight: 60

Console



```
Height entered: 170
Weight entered: 60
BMI: 20.761245674740486
BMI status: Normal
```

Uncontrolled Components

- To write an uncontrolled component, instead of writing an event handler for every state update, you can use a ref to get form values from the DOM
- Additionally, use the defaultValue property to specify initial value in React

```
<input defaultValue="Bob" type="text" ref={this.input} />
```

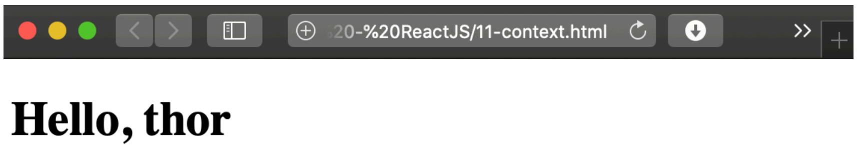
10.0 Context

- While passing `props` from parent to child, `...props` needs to be passed down through all the children
- Becomes tedious
- For example here, `App` needs to pass the props down to `DashBoard` and then finally to `Profile`, event though `DashBoard` does not use it

Inside the body `<script type="text/babel"></script>` tags

```
class Profile extends React.Component {
    render() {
        return (
            <h1>Hello, {this.props.uname} </h1>
        )
    }
}


function DashBoard(props) {
    return <Profile {...props}/>
}

class App extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return (
            <DashBoard {...this.props}/>
        );
    }
}


ReactDOM.render(
    <App uname="thor"/>,
    document.querySelector('#container')
);
```

Rendered in a browser



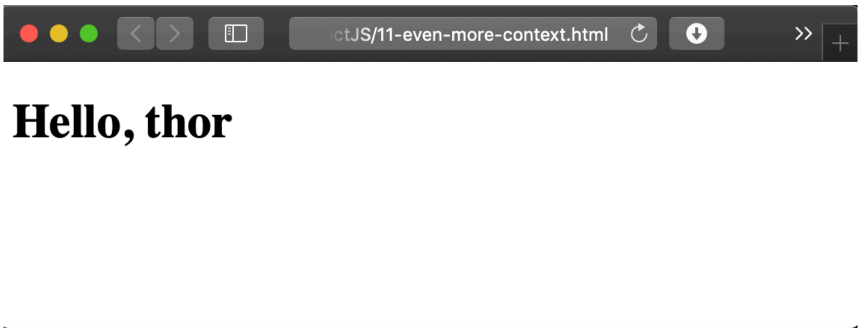
Hello, thor

- To make this using context, we create a context object by calling `React.createContext()`, which has two keys: `Provider` and `Consumer`
- The property (state) passes on from the `Provider` to the `Consumer`, not having to be passed through every level

Inside the body `<script type="text/babel"></script>` tags

```
const UContext = React.createContext();

class Profile extends React.Component {
    static contextType = UContext;
    render() {
        return (
            <h1>Hello, {this.context} </h1>
        );
    }
}


function DashBoard(props) {
    return <Profile/>
}

class App extends React.Component {
    render() {
        return (
            <UContext.Provider value={this.props.uname}>
                <DashBoard/>
            </UContext.Provider>
        );
    }
}


ReactDOM.render(
    <App uname="thor"/>,
    document.querySelector('#container')
);
```

Rendered in a browser



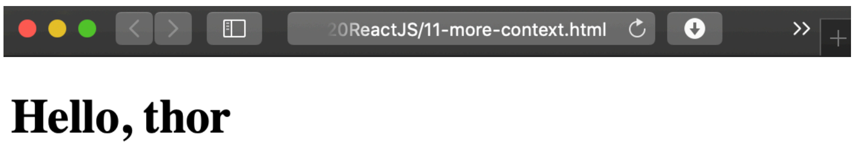
Hello, thor

Without defining a `static contextType`

Inside the body `<script type="text/babel"></script>` tags

```
const {Provider, Consumer} = React.createContext();

class Profile extends React.Component {
    render() {
        return (
            <Consumer>
                {function (user) {
                    return (
                        <h1>Hello, {user.uname} </h1>
                    )
                }}
            </Consumer>
        )
    }
}


function DashBoard(props) {
    return <Profile/>
}

class App extends React.Component {
    constructor(props) {
        super(props);

        this.state = {
            uname: ''
        }
    }

    componentDidMount() {
        this.setState({
            uname: this.props.uname
        })
    }

    render() {
        return (
            <Provider value={this.state}>
                <DashBoard/>
            </Provider>
        );
    }
}


ReactDOM.render(
    <App uname="thor"/>,
    document.querySelector('#container')
);
```

Rendered in a browser



Hello, thor