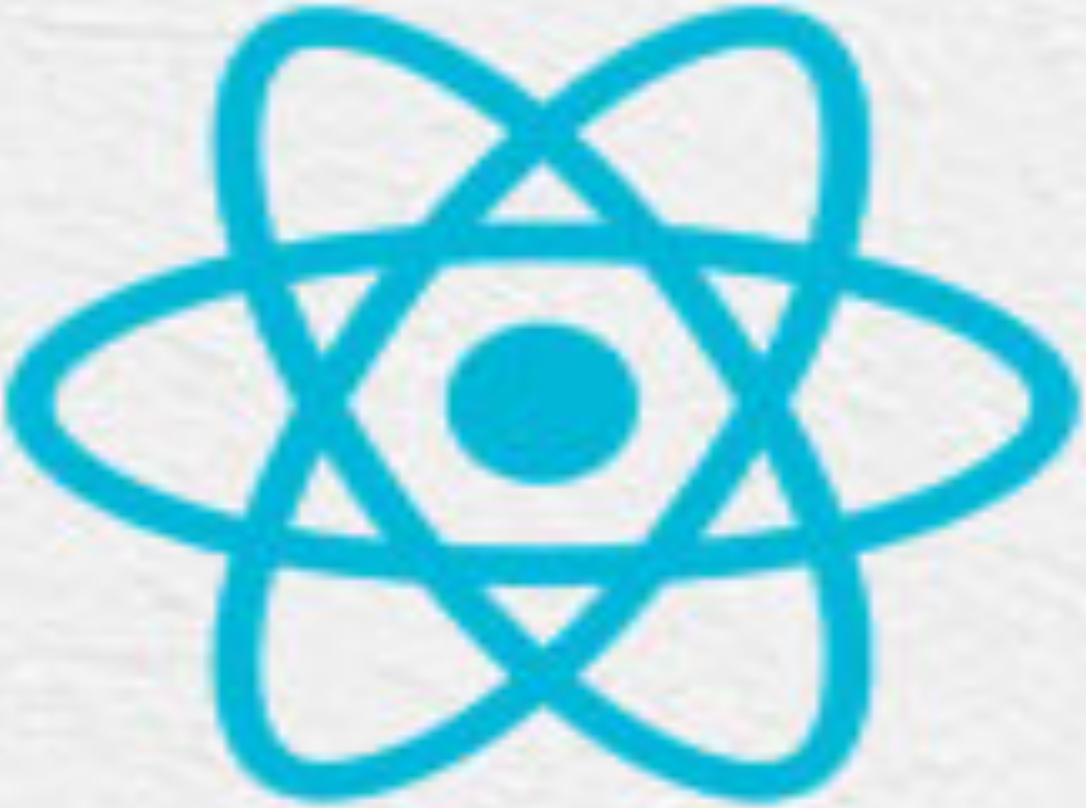


LEARN REACTJS

MERN Stack

Introduction

- The MERN stack is a popular stack of technologies for building a modern single-page application.

The MERN stack consists of the following technologies:

- **MongoDB**: A document-based open source database.
- **Express.js**: A web application framework for Node.js.
- **React.js**: A JavaScript front-end library for building user interfaces.
- **Node.js**: JavaScript run-time environment that executes JavaScript code outside of a browser (such as a server).

- React is a front-end library developed by Facebook. It is used for handling the view layer for web and mobile apps. ReactJS allows us to create reusable UI components. It is currently one of the most popular JavaScript libraries and has a strong foundation and large community behind it.

Advantages

- Automatic UI State Management
- Lightning-fast DOM Manipulation
- APIs to Create Truly Composable UIs
- Visuals Defined Entirely in JavaScript

- React allows you to (optionally) specify your visuals using an HTML-like syntax known as JSX that lives fully alongside your JavaScript. Instead of writing code to define your UI, you are basically specifying markup:

```
ReactDOM.render(
  <div>
    <h1>Batman</h1>
    <h1>Iron Man</h1>
    <h1>Nicolas Cage</h1>
</div>,
  destination
);
```

JAVASCRIPT

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

- JSX is a language that allows you to easily mix JavaScript and HTML-like tags to define user interface (UI) elements and their functionality. To build a web app using React, we need a way to take our JSX and convert it into plain old JavaScript that your browser can understand. We have two ways for that -

  - Set up a development environment around Node and a handful of build-tools. In this environment, every time you perform a build, all of your JSX is automatically converted into JS and placed on disk for you to reference like any plain JavaScript file.

  - Let your browser automatically convert JSX to JavaScript at runtime. You specify your JSX directly just like you would any old piece of JavaScript, and your browser takes care of the rest.

- Just below the title, add these two lines:

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

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

There is one more library we need to reference. Just below these two script tags, add the following line:

```
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
```

What we are doing here is adding a reference to the Babel JavaScript compiler. Babel does many cool things, but the one we care about is its ability to turn JSX into JavaScript.

MERN Stack

Example -1

```
<!DOCTYPE html>
<html>
<head>  <meta charset="utf-8">
 <title>React! React! React!</title>
 <script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>
 <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
 <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
</head>
<body>
 <script type="text/babel">
  ReactDOM.render(
  <h1>Welcome to React.js</h1>,
  document.body
 );
 </script>
</body>
</html>
```

Javascript


```
<script>   // don't have to use babel type

 ReactDOM.render(
    React.createElement("h1", null, "Welcome"),
            document.body
 );

 </script>
```

- The render method takes two arguments:
  - The HTML-like elements (aka JSX) you wish to output
  - The location in the DOM that React will render the JSX into
- **Changing the Destination**

```
<body>
  <div id="container"></div>
  <script type="text/babel">
    ReactDOM.render(
      <h1>Batman</h1>,
      document.querySelector(“#container”)
    );
  </script>
</body>
```

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

ReactDOM.render(
  <h1>Batman</h1>,
  destination
);
```

React.js - Components

- Components are one of the things that make React...well, React!
- Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and returns HTML via a render function.
- Components come in two types,
  - ❖Class components
  - ❖Function components

Eg –

```
ReactDOM.render(
  <div>
    <p>Hello, world!</p>
  </div>,
  document.querySelector("#container")
);
```

The above code is recreated by using a component .This is done by creating the class syntax

```
class HelloWorld extends React.Component {

}
```

This HelloWorld component is a component because it extends React.Component

React.js - Components

The render function inside a component is also responsible for dealing with JSX.

```
class HelloWorld extends React.Component {
   render() {
     return <p>Hello, React world!</p>;
   }
}
```

The way you use a component once you've defined it is by calling it, and we are going to call it from , the ReactDOM.render method.

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

React.js – Components Properties(Parameterized component)

We can add properties to a component in two parts

First Part: Updating the Component Definition

```
class HelloWorld extends React.Component {
   render() {
     return <p>Hello, {this.props.greetTarget}!</p>;
   }
}
```

property

The way you access a property is by referencing it via the this.props property that every component has access to. We place it inside curly brackets - { }. In JSX, if you want something to get evaluated as an expression, you need to wrap that something inside curly brackets. If you don't do that, the raw text this.props.greetTarget gets printed out.

Second Part: Modifying the Component Call

Once the component definition is updated , all that remains is to pass in the property value as part of the component call. That is done by adding an attribute with the same name as our property followed by the value you want to pass in.

```
ReactDOM.render(
  <div>
    <HelloWorld greetTarget="Batman"/>
    <HelloWorld greetTarget="Iron Man"/>
    <HelloWorld greetTarget="Catwoman"/>
  </div>,
  document.querySelector("#container")
);
```

React.js – Child Components

Your components can have children. The child elements in a component can be accessed by the children property accessed by this.props.children.

Eg –

```
class Buttonify extends React.Component {
  render() {
    return(
      <div>
        <button type={this.props.behavior}>{this.props.children}</button>
      </div>
    );
  }
}
```

React.js – Components Properties

The way you can use this component is by just calling it via the ReactDOM.render method

```
ReactDOM.render(
  <div>
    <Buttonify behavior="submit">SEND DATA</Buttonify>
  </div>,
  document.querySelector("#container")
);
```

We specify a custom property called behavior. This property allows us to specify the button element's type attribute, and it is accessed via this.props.behavior in the component definition's render method.

React.js – Styling Components

Using CSS to style our React content is actually straightforward React ends up spitting out regular HTML tags, except that there are just a few minor things to keep in mind.

Before you can use CSS, you need to first get a feel for what the HTML that React spits out is going to look. You can easily figure that out by looking the JSX defined inside the render methods.

Eg –

```
<style>
  #container {
    padding: 50px;
    background-color: #FFF;    }
 </style>
</head>
 <body>
 <div id="container"></div>
 </body>
```

React.js – Styling Components

```
<script type="text/babel">
 var destination = document.querySelector("#container");
  class Letter extends React.Component {
   render() {
     return(
       <div>
         {this.props.children}
       </div>
     );
   } }
  ReactDOM.render(
   <div>
     <Letter>A</Letter>
     <Letter>E</Letter>
     <Letter>I</Letter>
     <Letter>O</Letter>
     <Letter>U</Letter>
   </div>,
   destination
  );
</script>
```

React.js – Styling Components


```
... ▼<div id="container"> == $0
      ▼<div>
          <div>A</div>
          <div>E</div>
          <div>I</div>
          <div>O</div>
          <div>U</div>
        </div>
      </div>
```

DOM STRUCTURE

each individual vowel is wrapped inside its own set of div tags which is an HTML-ized expansion of the various JSX fragments

To affect our inner div elements, add the following inside our style tag:

```
div div div {
  padding: 10px;
  margin: 10px;
  background-color: #ffde00;
  color: #333;
  display: inline-block;
  font-family: monospace;
  font-size: 32px;
  text-align: center;
}
```

The div div div selector will ensure we style the right things but is too generic. In apps with more than three div elements (which will be very common), we may end up styling the wrong things.

React.js – Styling Components

The generic styling issue is solved by giving our inner div elements a class value . We designate the class value by using the className attribute instead of the class attribute. The reason has to do with the word class being a special keyword in JavaScript. After that Modify the CSS selector to target our div elements.

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

Modified css

```
.letter {
  padding: 10px;
  margin: 10px;
  background-color: #ffde00;
  color: #333;
  display: inline-block;
  font-family: monospace;
  font-size: 32px;
  text-align: center;
}
```

React.js – Creating style object

React favours an inline approach for styling content that doesn't use CSS as it is designed to help make your visuals more reusable. The goal is to have your components be little black boxes where everything related to how your UI looks and works gets stashed there.

The way you specify styles inside your component is by defining an object whose content is the CSS properties and their values. Once you have that object, you assign that object to the JSX elements you wish to style by using the style attribute.

React.js – Creating style object

```
class Letter extends React.Component {
  render() {
    var letterStyle = {
      padding: 10,
      margin: 10,
      backgroundColor: "#ffde00",
      color: "#333",
      display: "inline-block",
      fontFamily: "monospace",
      fontSize: 32,
      textAlign: "center"
    };
     return (
      <div style ={letterStyle} >
        {this.props.children}
      </div>
    );}}
```

Find the element we wish to apply the style on and set the style attribute to refer to that object.

React.js – Creating style object

Eg - Background Color Customizable

```
ReactDOM.render(
   <div>
      <Letter bgcolor="#58B3FF">A</Letter>
      <Letter bgcolor="#FF605F">E</Letter>
      <Letter bgcolor="#FFD52E">I</Letter>
      <Letter bgcolor="#49DD8E">O</Letter>
      <Letter bgcolor="#AE99FF">U</Letter>
   </div>,
   destination
);
```

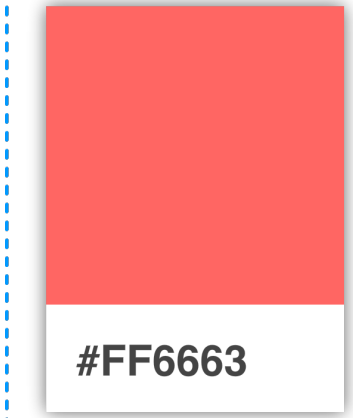
backgroundColor: this.props.bgcolor

In our letterStyle object, set the value of backgroundColor to this.props.bgColor to use the bgcolor property

React.js – Complex Components

Components are the primary ways through which React allows our visual elements to behave like little reusable bricks that contain all of the HTML, JavaScript and styling needed to run themselves. Beyond reusability, there is another major advantage components bring to the table. They allow for composability. You can combine components to create more complex components.

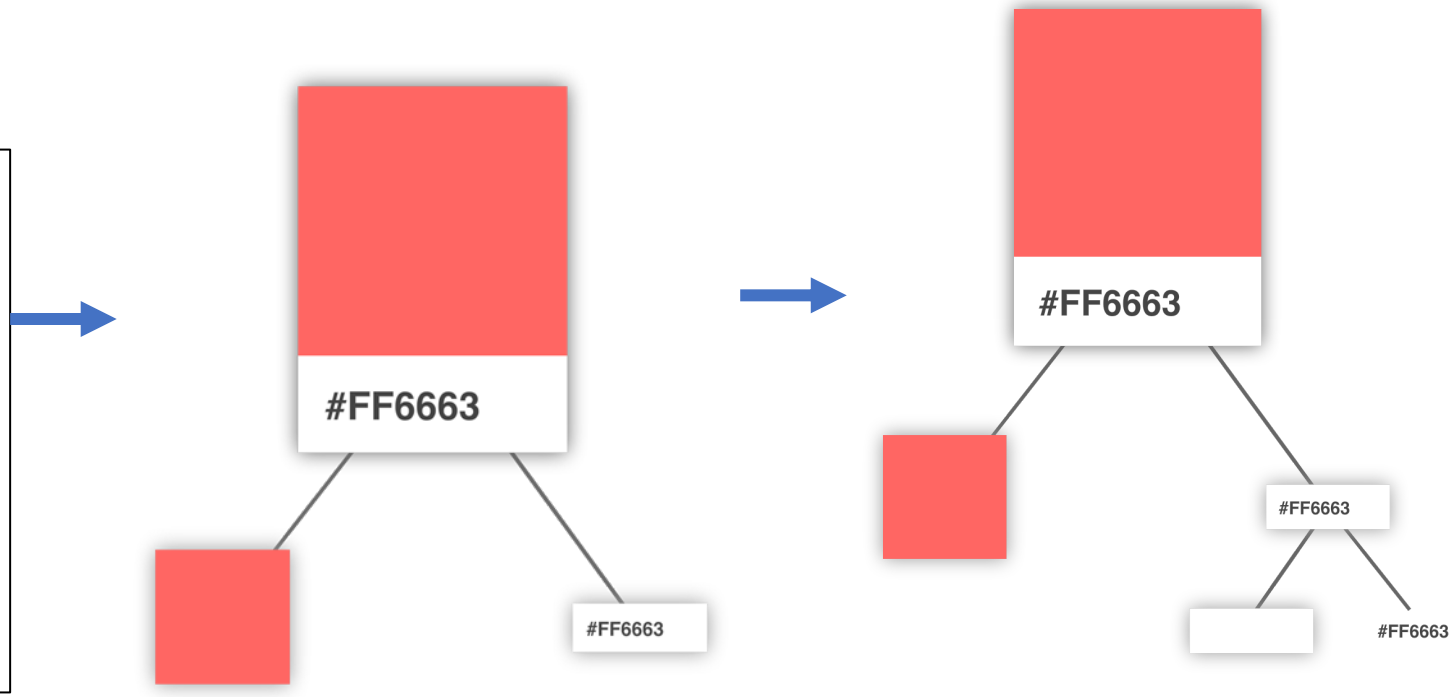
Eg - there are these small rectangular cards that help you match a color with a particular type of paint and lets re-create one of these cards using React.



Identifying the Major Visual Elements

The first step is to identify all of the visual elements we are dealing with. No visual element is too minor to omit - at least, not initially. The easiest way to start identifying the relevant pieces is to start with the obvious visual elements and then diving into the less obvious ones.

.
Arranging your visuals into this tree-like structure (aka a **visual hierarchy**) is a good way to get a better feel for how your visual elements are grouped. The goal of this exercise is to identify the important visual elements and break them into a parent/child arrangement until you can divide them no further.

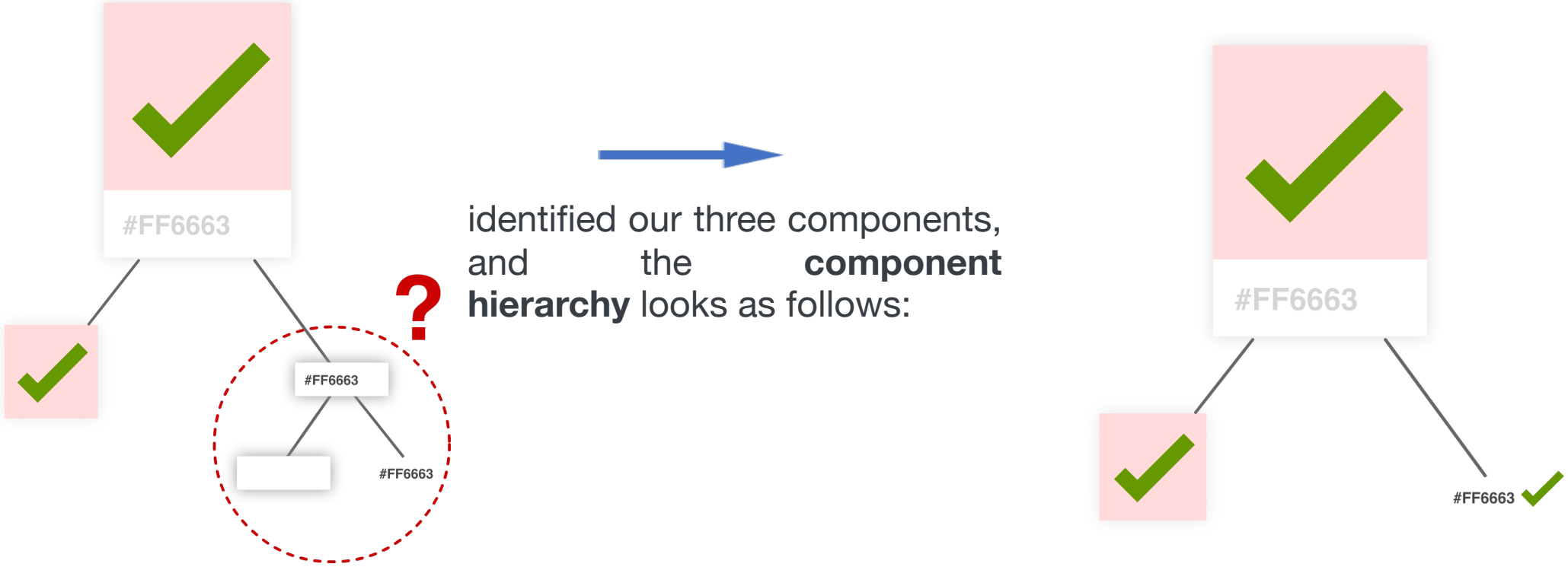


PES UNIVERSITY ONLINE

Identifying the Components

We need to figure out which of the visual elements we've identified will be turned into a component and which ones will not. The general rule is that our components should do just one thing. If you find that your potential component will end up doing too many things, you probably want to break your component into multiple components. On the flipside, if your potential component does too little, you probably want to skip making that visual element a component altogether

React.js – Complex Components



For figuring out which components to create, you should use the component hierarchy.

Creating the Components

it is time to define our three components. The names we will go with for our components will be Card, Label, and Square.

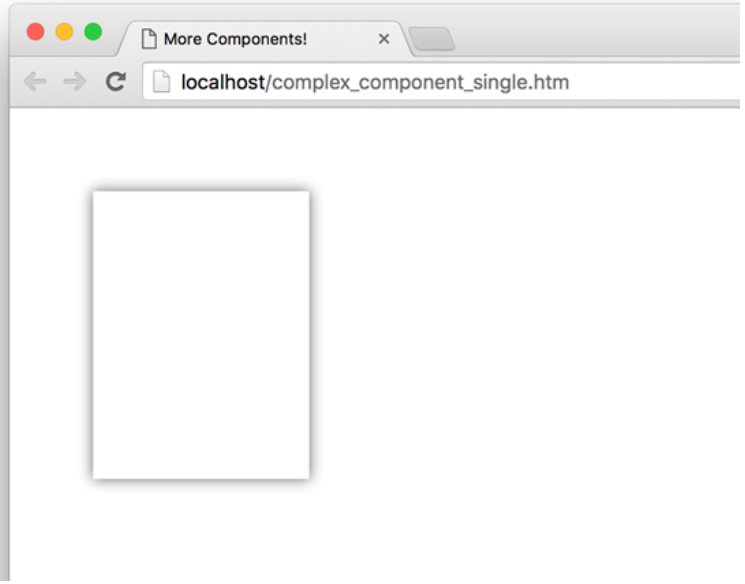
```
class Square extends React.Component {
  render() {
    return(
      <br/>
    );
  }
}
class Card extends React.Component {
  render() {
    return(
      <br/>
    );
  }
}
```

```
class Label extends React.Component {
  render() {
    return(
      <br/>
    );
  }
}
```

The Card Component

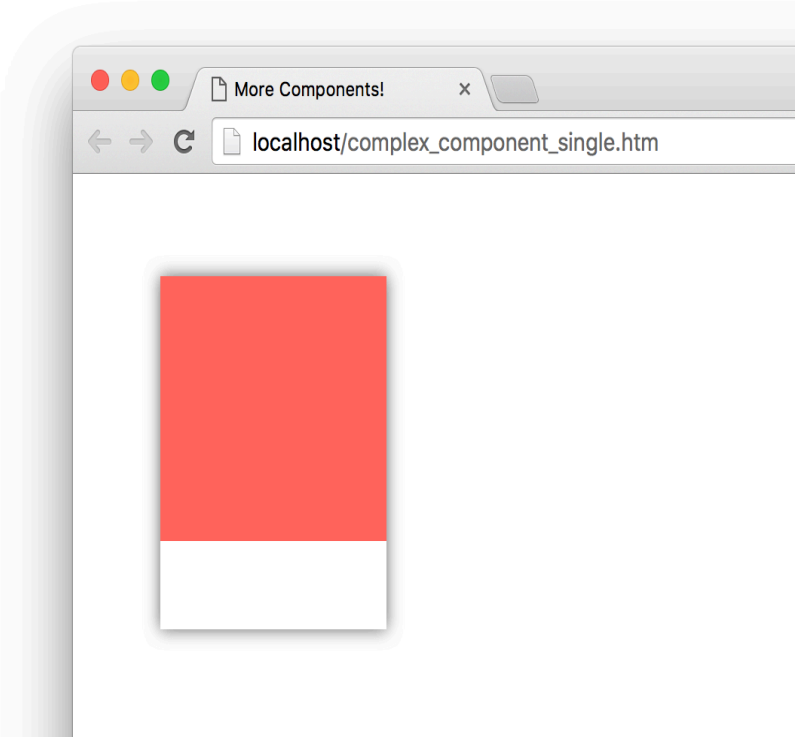
This component will act as the container that our Square and Label components will live in.

```
class Card extends React.Component {
  render() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      WebkitFilter: "drop-shadow(0px 0px 5px #666)",
      filter: "drop-shadow(0px 0px 5px #666)"
    };
    return (
        <div style={cardStyle}>
        </div>
    );
  }
}
```



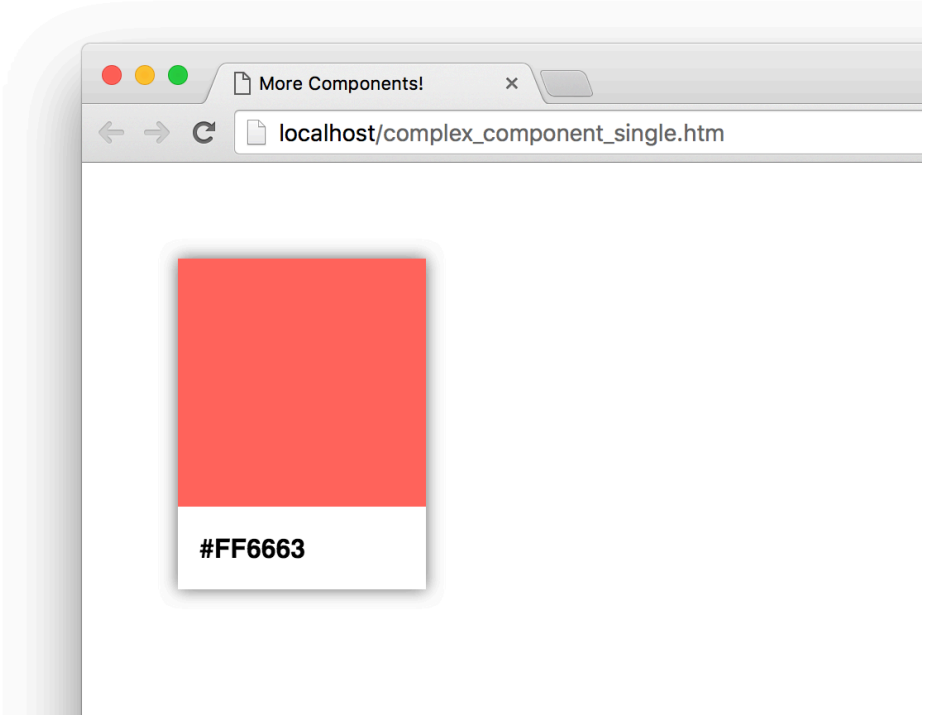
The Square Component

```
class Square extends React.Component {
  render() {
    var squareStyle = {
      height: 150,
      backgroundColor: "#FF6663"
    };

    return (
      <div style={squareStyle}>

      </div>
    );
  }
}
```



The Label Component

```
class Label extends React.Component {
  render() {
    var labelStyle = {
      fontFamily: "sans-serif",
      fontWeight: "bold",
      padding: 13,
      margin: 0
    };
     return (
      <p style={labelStyle}>#FF6663</p>
    );
  }
}
```



React.js – Stateful Components

- So far, we have seen static or stateless Components. They don't undergo state changes.
- Components need to change based on
  - User actions (clicks, keyboard inputs, etc.)
  - Other triggers (responses received from server, timers, etc.)
- Let's consider this Component that shows the number of seconds the user has been on the page

React.js – Stateful Components

We consider two components LightningCounter and LightningCounterDisplay

```
class LightningCounter extends React.Component {
    render() {
      return (
        <h1>Hello!</h1>
      );
    }
  }

ReactDOM.render(
    <LightningCounterDisplay/>,
    document.querySelector("#container")
  );
 </script>
</body>
</html>
```

```
class LightningCounterDisplay extends React.Component {
    render() {
      var divStyle = {
        width: 250,
        textAlign: "center",
        backgroundColor: "black",
        padding: 40,
        fontFamily: "sans-serif",
        color: "#999",
        borderRadius: 10
      };
      return (
        <div style={divStyle}>
          <LightningCounter/>
        </div>
      );
    }  }
```

PES UNIVERSITY ONLINE

Getting Our Counter On

A setInterval function that calls some code every 1000 milliseconds (aka 1 second). We will use the ***constructor*** method to initialize the counter

- The Component also has the method ***componentDidMount(React API)*** that can be used to start the counter
- It also exposes the ***setState (React API)*** method to update the state (the counter)

Setting the Initial State Value

We need a variable to act as our counter, and let's call this variable strikes which is part of our component's state. The way to do this is by creating a state object, making our strikes variable be a property of it, and ensure we set all of this up when our component is getting created.

```
class LightningCounter extends React.Component {

 constructor(props, context) {
    super(props, context);
     this.state = {
      strikes: 0
    };
  }
render() {
    return (
      <h1>{this.state.strikes}</h1>
    );
  }
}
```

We specify our state object inside our LightningCounter component's constructor.
This runs way before our component gets rendered, and what we are doing is telling React to set an object containing our strikes property (initialized to 0).

PES UNIVERSITY ONLINE

Starting Our Timer & Setting State

The timer should be going on and incrementing the strikes property. The setInterval function to increase the strikes property by 100 every second which is done when component has been rendered using the built-in componentDidMount method.

```
componentDidMount() {
   setInterval(this.timerTick, 1000);
 }
```

Starting Our Timer & Setting State

The timer should be going on and incrementing the strikes property. The setInterval function to increase the strikes property by 100 every second which is done when component has been rendered using the built-in componentDidMount method.

```
componentDidMount() {
   setInterval(this.timerTick, 1000);
 }
```

```
timerTick() {
   this.setState({
     strikes: this.state.strikes + 100
   });
 }
```

This object contains all the properties you want to merge into the state object. In our case, we are specifying the strikes property and setting its value to be 100 more than what it is currently.

React.js – Stateful Components

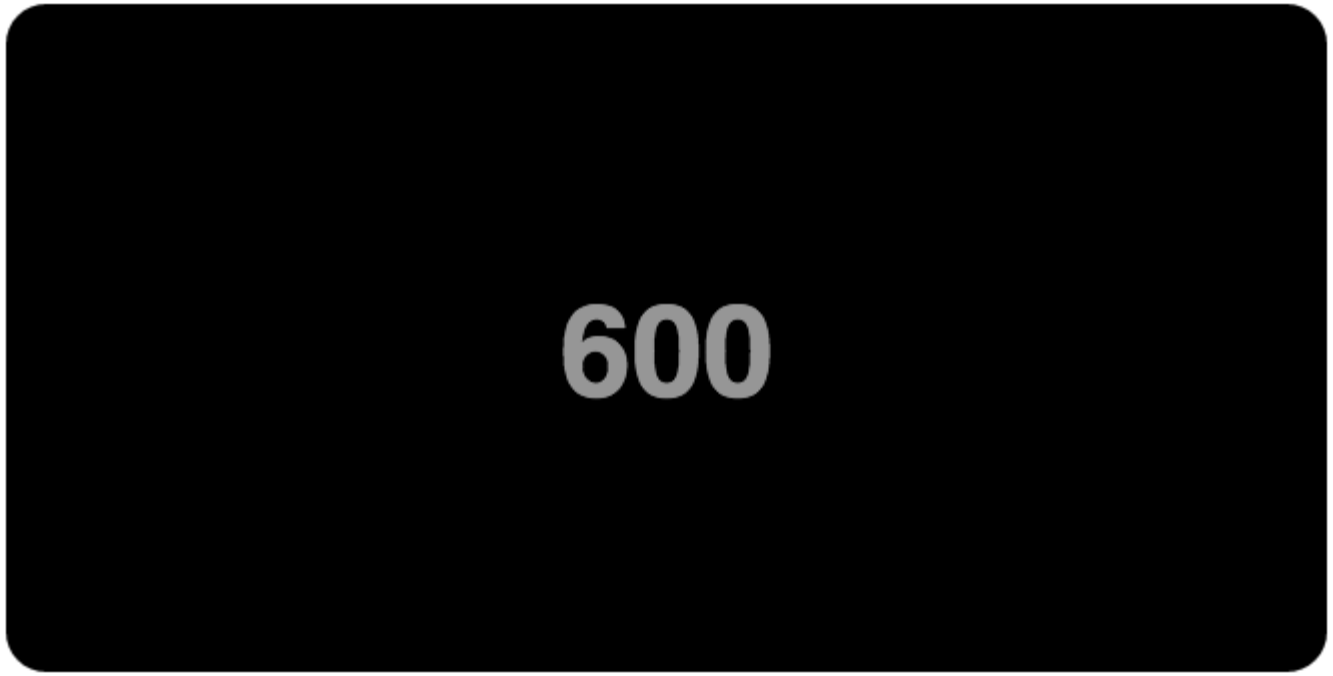
To bring in the stateful Component behaviour, that also ensures that state variable seconds is never out of sync

```
this.setState((prevState) => {  return {
strikes: prevState.strikes + 100
};
});
```

To ensure that the this.state works well in the timerTick method, add the following line to call that function with the context of the Component

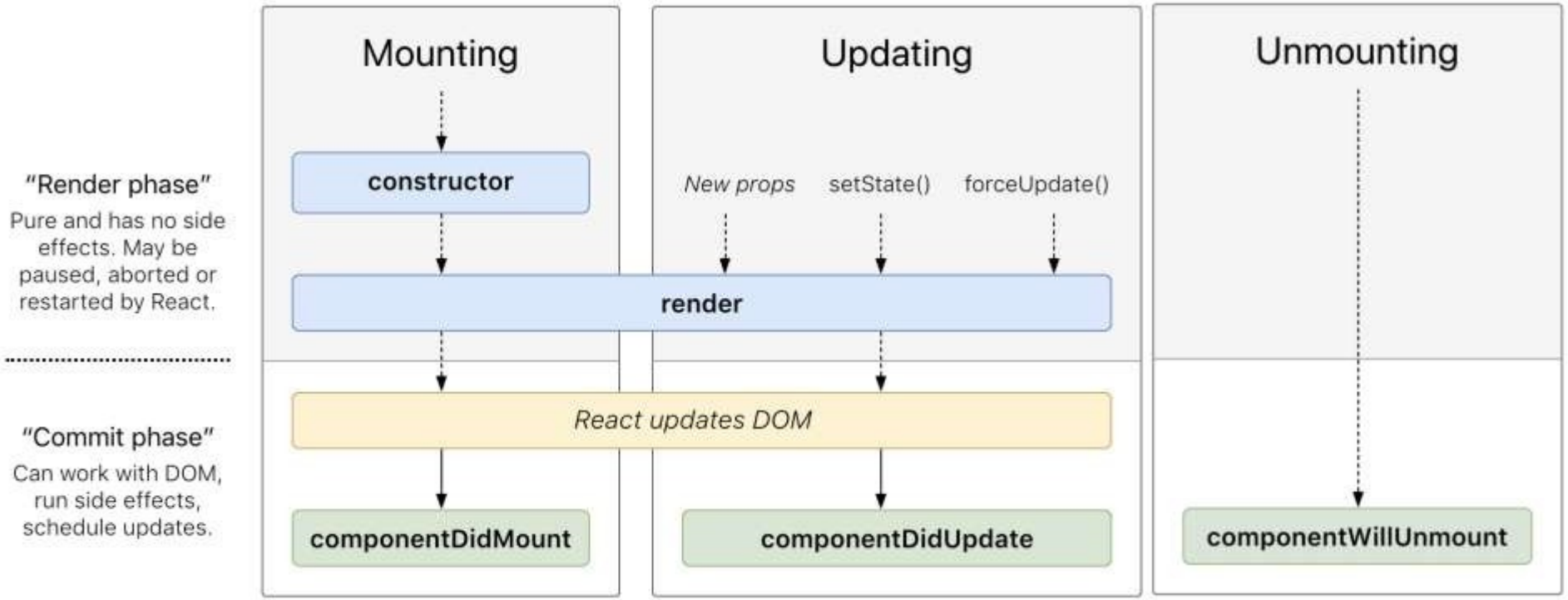
```
constructor(props, context) {
…
this.timerTick = this.timerTick.bind(this);
}
```

Whenever you call setState and update something in the state object, your component's render method gets automatically called. This kicks of a cascade of render calls for any component whose output is also affected.



React.js – Component Life Cycle

Components undergo these following life cycle states and their associated events



componentDidMount()

componentDidMount() is invoked immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request. The setState() is called immediately in componentDidMount(). It will trigger an extra rendering, but it will happen before the browser updates the screen.

componentDidUpdate()

componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render.

It is used to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props

componentWillUnmount()

componentWillUnmount() is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in componentDidMount().

You should not call setState() in componentWillUnmount() because the component will never be re-rendered. Once a component instance is unmounted, it will never be mounted again.

```
if (this.state.strikes == 500) {


ReactDOM.unmountComponentAtNode(document.querySelector("#container")
);
  }
 }
 componentWillUnmount() {
  console.log("Component is about to be unmounted!");
    clearInterval(this.timer);
 }
```

React.js – Stateless component

Let's consider a Component as follows

```
class Stuff extends React.Component {
render() {
      return (
      <p>Batman</p>,
      );
      }
};
```

The component is just returning a JSX element without any constructor, state or need for Component Life Cycle methods like ComponentDidMount

React.js – Stateless component

Such Stateless components can be written as functions that just return the

JSX element as follows

```
function Stuff () {  return (
<p>Batman</p>,
);
};
```

The code to render such Components remain the same:

```
ReactDOM.render(<Stuff/>, destination )
```

React.js – Stateless component

Properties are passed to these function components like regular parameters

```
function Stuff (props) {
return (
    <p>{props.name}</p>,
    );
    };
```

The properties are specified as before:

```
ReactDOM.render(<Stuff name="Batman"/>, destination )
```

React.js – Keys

- When returning an array or list of elements, the individual element should be uniquely identified by a key property
- The key property helps React identify each element in the list for rendering or updating etc.

- Not specifying the key property will display a warning on the console:

Warning: Each child in an array or iterator should have a unique "key" prop.

React.js – Keys

When returning a list of elements:

```
function Stuff () {
  return (
  [
    <p>Batman</p>,
    <p>Ironman</p>,
    <p>Spiderman</p>
  ]
  );
};
```

Specify key property as follows:

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

To Recap, the map method on an array or list, calls a callback function for each element of the array. The map() creates a new array with the results of calling a provided function on every element in the calling array.

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
// expected output [2,4,6,8,10]
```

React.js – Keys

We return a <li> element for each item.

```
const numbers = [1, 2, 3, 4, 5];

const listItems = numbers.map((number) => <li>{number}</li>);
```

We include the entire listItems array inside a <ul> element, and render it to the DOM:

```
ReactDOM.render(
<ul>{listItems}</ul>,document.getElementById('root') );
```

The code will result in error, as the list of items does not have the key property.

React.js – Keys

By assigning the key property to the list items

```
function NumberList(props) {

const numbers = props.numbers;
const listItems = numbers.map((number) =>
<li key={number.toString()}> {number} </li>
);
      return ( <ul>{listItems}</ul> );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
<NumberList numbers={numbers} />,  document.getElementById('root') );
```

React.js – Refs and passing refs

React Refs are a useful feature that act as a means to reference a DOM element or a class component from within a parent component. This then give us a means to read and modify that element.

Using refs give us a way to access elements while bypassing state updates and re-renders. Refs also provide some flexibility for referencing elements within a child component from a parent component, in the form of *ref forwarding.*

Refs are usually defined in the constructor of class components, or as variables at the top level of functional components, and then attached to an element in the render() function.

React.js – Refs and passing refs

- Refs can be used to induce changes in Components or Elements after they have been rendered (like animation, user action events etc.)
- To create a ref add a property which is set to a callback function that sets a reference to that element

```
<input type="text" ref={this.setTextInputRef} />
 this.setTextInputRef = element=> {
this.textInput = element;
            };
```

```
if (this.textInput) {
 this.textInput.focus();
console.log(this.textInput.value)
}
```

Here ***element*** refers to the input text element. Now this.textInput is the reference to that element and can be used to perform raw DOM operations

React.js – Refs and passing refs

Parent element can pass a ref callback to its child element, to get reference to the child element

```
function CustomTextInput(props) {
 return (
      <div>
      <input ref={props.inputRef} />
      </div>
);
}
class Parent extends React.Component {  render() {
return (
<CustomTextInput inputRef={el =>  this.inputElement = el} />
);
}
}
```

React.js – Refs and passing refs

Refs are created when a component is rendered and can be defined either in the componentDidMount() or in the constructor().

Refs can be created using **React.createRef()** method. and attach this to our input element using the ref attribute.

When the ref attribute is used on an HTML element, the ref created in the constructor with React.createRef() receives the underlying DOM element as its current property.

- When the ref attribute is used on a custom class component, the ref object receives the instance of the component as its current and through we can access the props and the methods of that component.

React.js – Refs and passing refs

```
constructor(props)
   {
   super(props);
   this.quantityref=React.createRef();   // This will create a reference object
   }
```

React.js – Events

- React Events are very similar to DOM Events, with some syntax differences
  - React events are named using camelCase, rather than lowercase
  - With JSX you pass a function as the event handler, rather than a string
- The event object passed to the event handlers are SyntheticEvent object, with the following characteristics
  - SyntheticEvent is a wrapper around the DOMEvent object
  - The 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, use e.preventDefault() or e.stopPropogation()

Eg - Listening and Reacting to Events

```
class Counter extends React.Component {
  render() {
    var textStyle = {……
};

    return (
      <div style={textStyle}>
        {this.props.display}
      </div>
    );
  }
}
```

React.js – Events

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

    this.state = {
      count: 0
    };
  }

  render() {
    var backgroundStyle = {……
};
```

```
var buttonStyle = {……
};

    return (
      <div style={backgroundStyle}>
        <Counter display={this.state.count} />
        <button style={buttonStyle}>+</button>
      </div>
    );
  }
}

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

PES UNIVERSITY ONLINE

Making the Button Click Do Something

Each time we click on the plus button, the value of our counter has to increase by one.

- Listen for the click event on the button.
- Implement the event handler where we react to the click and increase the value of our this.state.count property that our counter relies on.

In React, we can listen to an event by specifying everything inline in our JSX itself. More specifically, you specify both the event you are listening for and the event handler that will get called all inside your markup in the return function inside our CounterParent component,

```
return (
  <div style={backgroundStyle}>
    <Counter display={this.state.count}/>
    <button onClick={this.increase} style={buttonStyle}>+</button>
  </div>
);
```

The increase function is defined Inside our CounterParent component,

```
increase(e) {
  this.setState({
    count: this.state.count + 1
  });
 }
```

React.JS – Events
Synthetic Event Object

- The Synthetic Event Object has the following properties
  - boolean bubbles
  - boolean cancelable
  - DOMEventTarget currentTarget
  - boolean defaultPrevented
  - number eventPhase
  - boolean isTrusted
  - DOMEvent nativeEvent
  - void preventDefault()
  - boolean isDefaultPrevented()
  - void stopPropagation()
  - boolean isPropagationStopped()
  - void persist()
  - DOMEventTarget target
  - number timeStamp
  - string type

In React, when an event is specified , we are not directly dealing with regular DOM events. Instead, we are dealing with a React-specific event type known as a SyntheticEvent. Your event handlers don't get native event arguments of type MouseEvent, KeyboardEvent, etc. They always get event arguments of type SyntheticEvent that wrap your browser's native event instead.

React.JS – Events
Synthetic Event Object

- A SyntheticEvent that wraps a KeyboardEvent will have access to these additional keyboard-related properties:

    - boolean altKey
    - number charCode
    - boolean ctrlKey
    - boolean getModifierState(key)
    - string key
    - number keyCode
    - string locale
    - number location
    - boolean repeat
    - boolean shiftKey

React.JS – Forms

Introduction

- Two main functionalities associated with any form is when
  - input values are changed (using ***onChange*** event)
  - form is submitted (using ***onSubmit*** event)

- In HTML, form elements such as <input>, <textarea>, and <select> typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with setState().

- An input form element whose value is controlled by React in this way is called a **"controlled component"**

React.JS – Forms
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 (by setState)
- The value is also set to the state property to keep it updated at all times (updated by React) – this is termed as ***"single source of truth"***

React.JS – Forms
Controlled Components

Eg –

```
<textarea> Default Value </textarea>
```

can be changed to

```
<textarea value={this.state.value} />
```

Eg -

```
<select>
    <option value="grapefruit">Grapefruit</option>
    <option selected value="coconut"> Coconut</option>
</select>
```

can be changed to

```
<select value={this.state.value}>
    <option value="grapefruit">Grapefruit</option>
    <option  value="coconut"> Coconut</option>
</select>
```

```
this.state = {value: 'coconut'};
```

Will be defined in the constructor

React.JS – Forms Handling
Multiple Inputs

- To handle multiple inputs by writing a common change handler as follows

```
handleChange(event) {
    name = event.target.name;
    value = event.target.value;
    this.setState({
        [name]: value
    });
```

where [name] is the notation for computed property name

Note: setState can be called only the property that is changed

React.JS – Forms
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.
- This we have already seen in the previous lessons
- Additionally, use the defaultValue property to specify initial value in React

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

React.JS – Forms
Context

In a typical React application, data is passed top-down (parent to child) via props, but this can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application.

Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree.

When to Use Context

Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language.

React.JS – Forms

Context

```
class App extends React.Component {
  render() {
    return <Toolbar theme="dark" />;
  }
}

function Toolbar(props) {
return (
    <div>
      <ThemedButton theme={props.theme} />
    </div>
  );}

class ThemedButton extends React.Component {
  render() {
    return <Button theme={this.props.theme} />;
  }}
```

The Toolbar component must take an extra "theme" prop and pass it to the ThemedButton. This can become painful if every single button in the app needs to know the theme because it would have to be passed through all components.

React.JS – Forms
Context

```
const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
return (
     <ThemeContext.Provider value="dark">
       <Toolbar />
     </ThemeContext.Provider>
   );
  }
}

function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}
```

```
class ThemedButton extends React.Component {
static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```

Use a Provider to pass the current theme to the tree below. Any component can read it, no matter how deep it is.

React will find the closest theme Provider above and use its value.