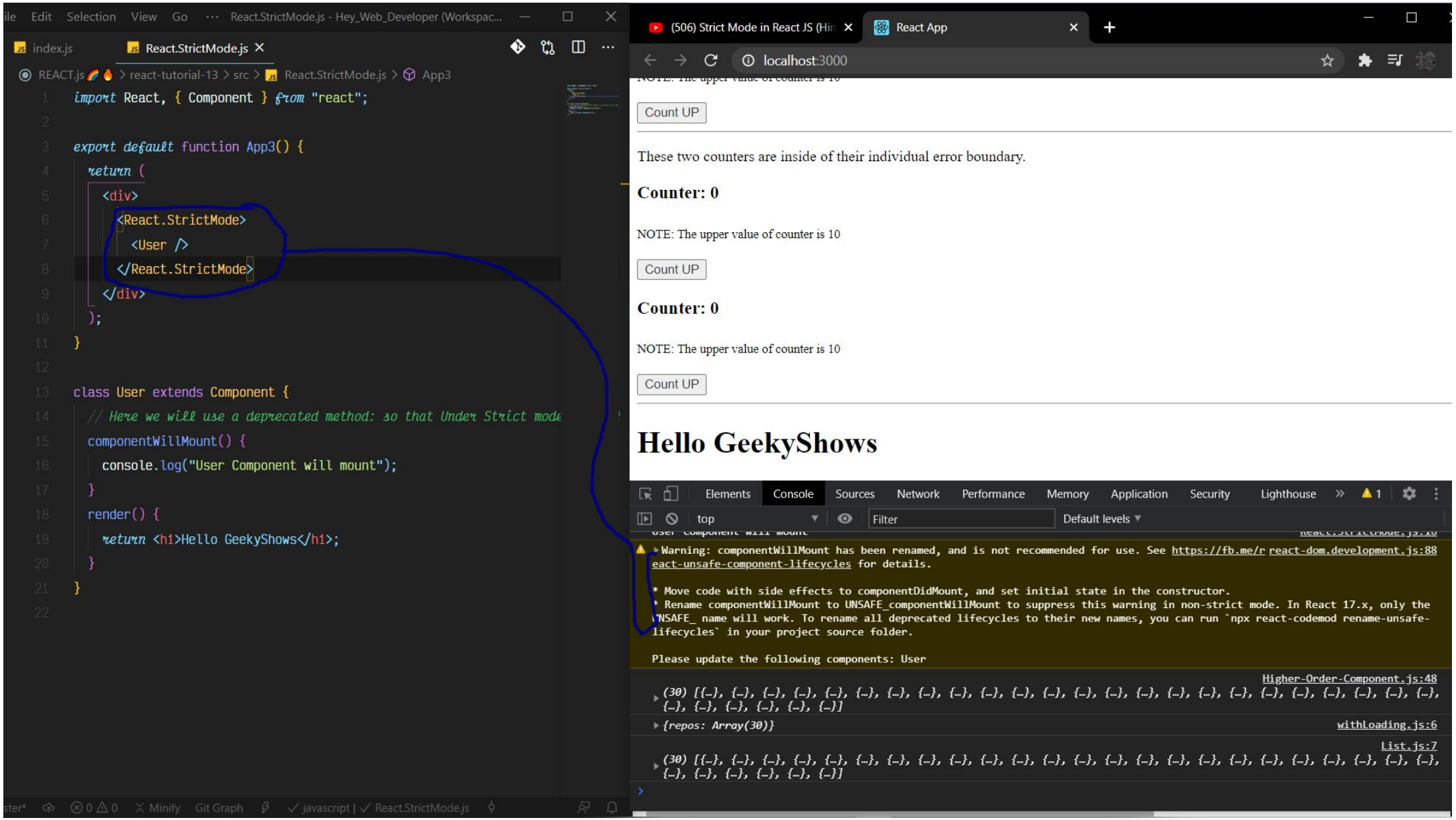
- *Update Yourself*
- *Build Application*
- *JSON*
- *Ajax*
- Routing - *React-Router*
- Design - *React-Bootstrap*
- Data Fetching (http request GET & POST) – *Axios*
- State Management – *Redux*


File Edit Selection View Go ··· React.StrictMode.js - Hey_Web_Developer (Workspac...
index.js
React.StrictMode.js
REACT.js 🌈🔥 > react-tutorial-13 > src > React.StrictMode.js > App3
1 import React, { Component } from "react";
2
3 export default function App3() {
4 return (
5
6 <React.StrictMode>
7 <User />
8 </React.StrictMode>
9
10);
11 }
12
13 class User extends Component {
14 // Here we will use a deprecated method: so that Under Strict mode
15 componentWillMount() {
16 console.log("User Component will mount");
17 }
18 render() {
19 return <h1>Hello GeekyShows</h1>;
20 }
21 }
22
ster* ⊗ 0 ⚠ 0 Minify Git Graph ✓ javascript | ✓ React.StrictMode.js
(506) Strict Mode in React JS (Hin
React App
localhost:3000
Count UP
These two counters are inside of their individual error boundary.
Counter: 0
NOTE: The upper value of counter is 10
Count UP
Counter: 0
NOTE: The upper value of counter is 10
Count UP
Hello GeekyShows
Elements Console Sources Network Performance Memory Application Security Lighthouse
top
Filter
Default levels
Warning: componentWillMount has been renamed, and is not recommended for use. See https://fb.me/react-unsafe-component-lifecycles for details.
react-dom.development.js:88
* Move code with side effects to componentDidMount, and set initial state in the constructor.
* Rename componentWillMount to UNSAFE_componentWillMount to suppress this warning in non-strict mode. In React 17.x, only the UNSAFE_ name will work. To rename all deprecated lifecycles to their new names, you can run `npx react-codemod rename-unsafe-lifecycles` in your project source folder.
Please update the following components: User
Higher-Order-Component.js:48
(30) [{…}, {…}]
{repos: Array(30)}
withLoading.js:6
List.js:7
(30) [{…}, {…}]

Strict Mode

StrictMode currently helps with:

- Identifying components with unsafe lifecycles
- Warning about legacy string ref API usage
- Warning about deprecated findDOMNode usage
- Detecting unexpected side effects
- Detecting legacy context API

Strict Mode

StrictMode is a tool for highlighting potential problems in an application.

StrictMode does not render any visible UI. It activates additional checks and warnings for its descendants.

Strict mode checks are run in development mode only; they do not impact the production build.

Ex:-

```
<React.StrictMode>
    <User />
</React.StrictMode>
```

Error boundary in class

A class component can becomes an error boundary if it defines a new lifecycle methods either static getDerivedStateFromError() or componentDidCatch(error, info). We can use static getDerivedStateFromError() to render a fallback UI when an error has been thrown, and can use componentDidCatch() to log error information.

An error boundary can?t catch the error within itself. If the error boundary fails to render the error message, the error will go to the closest error boundary above it. It is similar to the catch {} block in JavaScript.

React Error Boundaries

In the past, if we get any JavaScript errors inside components, it corrupts the React?s internal state and put React in a broken state on next renders. There are no ways to handle these errors in React components, nor it provides any methods to recover from them. But, **React 16** introduces a new concept to handle the errors by using the **error boundaries**. Now, if any JavaScript error found in a part of the UI, it does not break the whole app.

Error boundaries are React components which catch JavaScript errors anywhere in our app, log those errors, and display a fallback UI. It does not break the whole app component tree and only renders the fallback UI whenever an error occurred in a component. Error boundaries catch errors during rendering in component lifecycle methods, and constructors of the whole tree below them.

Note:

Sometimes, it is not possible to catch Error boundaries in React application. These are:

- Event handlers
- Asynchronous code (e.g. setTimeout or requestAnimationFrame callbacks)
- Server-side rendering
- Errors are thrown in the error boundary itself rather than its children.

componentDidCatch()

This lifecycle method is invoked after an error has been thrown by a descendant component.
Use componentDidCatch() to log error information.

Syntax:-

```
componentDidCatch(error, info) {

}
```

Where,

error - The error that was thrown.

info - An object with a componentStack key containing information about which component threw the error.

static getDerivedStateFromError()

This lifecycle method is invoked after an error has been thrown by a descendant component. It receives the error that was thrown as a parameter and should return a value to update state.

Use static getDerivedStateFromError() to render a fallback UI after an error has been thrown.

Syntax:-

```
static getDerivedStateFromError(error){
```

Error Boundaries

Error boundaries do not catch errors for:

- Event handlers
- Asynchronous code (e.g. setTimeout or requestAnimationFrame callbacks)
- Server side rendering
- Errors thrown in the error boundary itself (rather than its children)

Error Boundaries

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods *static getDerivedStateFromError()* or *componentDidCatch()*.

PROVIDE A COMPONENT WITH ANY PROP YOU WANT

This is a popular use case for HOCs. We can study our code base and note what reusable prop is needed across components. Then, we can have a wrapper HOC to provide those components with the reusable prop.

Let's use the example above:

```
// A simple component
const HelloComponent = ({ name, ...otherProps }) => (
 <div {...otherProps}>Hello {name}!</div>
);
```

Let's create a HOC named `withNameChange` that sets a `name` prop on a base component to "New Name".

```
const withNameChange = (BaseComponent) => (props) => (
  <BaseComponent {...props} name='New Name' />
);
```

In order to use the HOC on our `HelloComponent`, we wrap the HOC around the component, create a pure component named `EnhancedHello2`, and assign the HOC and our `HelloComponent` like so:

```
const EnhancedHello2 = withNameChange(HelloComponent);
```

To make a change to our `HelloComponent`, we can render the `EnhancedHello` component like so:

```
<EnhancedHello />
```

Now, the text in our `HelloComponent` becomes this:

```
<div>Hello New World</div>
```

To change the `name` prop, all we have to do is this:

```
<EnhancedHello name='Shedrack' />
```

The text in our `HelloComponent` becomes this:

```
<div>Hello Shedrack</div>
```

PROVIDE COMPONENTS WITH SPECIFIC STYLING

Continuing the use case above, based on whatever UI state you get from the HOC, you can render specific styles for specific UI states. For example, if the need arises in multiple places for styles like `backgroundColor`, `fontSize` and so on, they can be provided via a HOC by wrapping the component with one that just injects props with the specific `className`.

Take a very simple component that renders "hello" and the name of a person. It takes a `name` prop and some other prop that can affect the rendered JavaScript XML (JSX).

```
// A simple component
const HelloComponent = ({ name, ...otherProps }) => (
 <div {...otherProps}>Hello {name}!/div>
);
```

Let's create a HOC named `withStyling` that adds some styling to the "hello" text.

```
const withStyling = (BaseComponent) => (props) => (
  <BaseComponent {...props} style={{ fontWeight: 700, color: 'green' }} />
);
```

In order to make use of the HOC on our `HelloComponent`, we wrap the HOC around the component. We create a pure component, named `EnhancedHello`, and assign the HOC and our `HelloComponent`, like so :

```
const EnhancedHello = withStyling(HelloComponent);
```

To make a change to our `HelloComponent`, we render the `EnhancedHello` component:

```
<EnhancedHello name='World' />
```

Now, the text in our `HelloComponent` becomes this:

```
<div style={{fontWeight: 700, color: 'green' }}>Hello World</div>
```

CONDITIONALLY RENDER COMPONENTS

Suppose we have a component that needs to be rendered only when a user is authenticated — it is a protected component. We can create a HOC named `WithAuth()` to wrap that protected component, and then do a check in the HOC that will render only that particular component if the user has been authenticated.

A basic `withAuth()` HOC, according to the example above, can be written as follows:

```
// withAuth.js
import React from "react";
export function withAuth(Component) {
    return class AuthenticatedComponent extends React.Component {
        isAuthenticated() {
            return this.props.isAuthenticated;
        }

        /**
         * Render
         */
        render() {
            const loginErrorMessage = (
                <div>
                    Please <a href="/login">login</a> in order to view this part of the a
                </div>
            );

            return (
                <div>
                    { this.isAuthenticated === true ? <Component {...this.props} /> : log
                </div>
            );
        }
    };
}

export default withAuth;
```

The code above is a HOC named `withAuth`. It basically takes a component and returns a new component, named `AuthenticatedComponent`, that checks whether the user is authenticated. If the user is not authenticated, it returns the `loginErrorMessage` component; if the user is authenticated, it returns the wrapped component.

Note: `this.props.isAuthenticated` has to be set from your application's logic. (Or else use react-redux to retrieve it from the global state.)

To make use of our HOC in a protected component, we'd use it like so:

```
// MyProtectedComponent.js
import React from "react";
import {withAuth} from "./withAuth.js";

export class MyProectedComponent extends React.Component {
    /**
     * Render
     */
    render() {
        return (
            <div>
                This is only viewable  by authenticated users.
            </div>
        );
    }
}

// Now wrap MyPrivateComponent with the requireAuthentication function
export default withAuth(MyPrivateComponent);
```

Here, we create a component that is viewable only by users who are authenticated. We wrap that component in our `withAuth` HOC to protect the component from users who are not authenticated.

HIGHER-ORDER COMPONENTS (HOCS) IN REACT WERE INSPIRED BY higher-order functions in JavaScript. A HOC is an advanced technique for reusing logic in React components. It is a pattern created out of React's compositional nature.

HOCs basically incorporate the don't-repeat-yourself (DRY) principle of programming, which you've most likely come across at some point in your career as a software developer. It is one of the best-known principles of software development, and observing it is very important when building an application or writing code in general.

Copy

```
import React from 'react';
// Take in a component as argument WrappedComponent
const higherOrderComponent = (WrappedComponent) => {
// And return another component
  class HOC extends React.Component {
    render() {
      return <WrappedComponent />;
    }
  }
  return HOC;
};
```

Structure Of A Higher-Order Component

A HOC is structured like a higher-order function:

- It is a component.
- It takes another component as an argument.
- Then, it returns a new component.
- The component it returns can render the original component that was passed to it.

Now, we can understand the **working of HOCs** from the below example.

```
//Function Creation
function add (a, b) {
  return a + b
}
function higherOrder(a, addReference) {
  return addReference(a, 20)
}
//Function call
higherOrder(30, add) // 50
```

In the above example, we have created two functions **add()** and **higherOrder()**. Now, we provide the add() function as an **argument** to the higherOrder() function. For invoking, rename it **addReference** in the higherOrder() function, and then **invoke it**.

Here, the function you are passing is called a callback function, and the function where you are passing the callback function is called a **higher-order(HOCs)** function.

Higher-Order Components

A higher-order component (HOC) is an advanced technique in React for reusing component logic. HOCs are not part of the React API, per se. They are a pattern that emerges from React's compositional nature.

Concretely, **a higher-order component is a function that takes a component and returns a new component**.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

Whereas a component transforms props into UI, a higher-order component transforms a component into another component.

HOCs are common in third-party React libraries, such as Redux's `connect` and Relay's `createFragmentContainer`.

Higher Order Components

A Higher-Order Component (HOC) is an advanced technique in React for reusing component logic.

HOCs are common in third-party React libraries.

A HOC is a function that takes a component and returns a new component.

Syntax:-

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

Ex:-

```
const FacebookJob = withLanguage(ReactJS)
const Army = withArm(Men) { training }
const Army = (Men) => { training }
```

Consuming Multiple Contexts

To keep context re-rendering fast, React needs to make each context consumer a separate node in the tree.

```
// Theme context, default to light theme
const ThemeContext = React.createContext('light');

// Signed-in user context
const UserContext = React.createContext({
  name: 'Guest',
});

class App extends React.Component {
  render() {
    const {signedInUser, theme} = this.props;

    // App component that provides initial context values
    return (
      <ThemeContext.Provider value={theme}>
        <UserContext.Provider value={signedInUser}>
          <Layout />
```


```
function Layout() {
  return (
    <div>
      <Sidebar />
      <Content />
    </div>
  );
}

// A component may consume multiple contexts
function Content() {
  return (
    <ThemeContext.Consumer>
      {theme => (
        <UserContext.Consumer>
          {user => (
            <ProfilePage user={user} theme={theme} />
          )}
        </UserContext.Consumer>
      )}
    </ThemeContext.Consumer>
  );
}
```

If two or more context values are often used together, you might want to consider creating your own render prop component that provides both.

Context.displayName

Context object accepts a `displayName` string property. React DevTools uses this string to determine what to display for the context.

For example, the following component will appear as MyDisplayName in the DevTools:

```
const MyContext = React.createContext(/* some value */);
MyContext.displayName = 'MyDisplayName';

<MyContext.Provider> // "MyDisplayName.Provider" in DevTools
<MyContext.Consumer> // "MyDisplayName.Consumer" in DevTools
```

Context.Consumer

```
<MyContext.Consumer>
  {value => /* render something based on the context value */}
</MyContext.Consumer>
```

A React component that subscribes to context changes. This lets you subscribe to a context within a function component.

Requires a function as a child. The function receives the current context value and returns a React node. The `value` argument passed to the function will be equal to the `value` prop of the closest Provider for this context above in the tree. If there is no Provider for this context above, the `value` argument will be equal to the `defaultValue` that was passed to `createContext()`.

The `contextType` property on a class can be assigned a Context object created by `React.createContext()`. This lets you consume the nearest current value of that Context type using `this.context`. You can reference this in any of the lifecycle methods including the render function.

> **Note:**
> You can only subscribe to a single context using this API. If you need to read more than one see Consuming Multiple Contexts.
>
> If you are using the experimental public class fields syntax, you can use a **static** class field to initialize your `contextType`.

```
class MyClass extends React.Component {
  static contextType = MyContext;
  render() {
    let value = this.context;
    /* render something based on the value */
  }
}
```

Class.contextType

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* perform a side-effect at mount using the value of MyContext */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /* render something based on the value of MyContext */
  }
}
MyClass.contextType = MyContext;
```

The `contextType` property on a class can be assigned a Context object created by `React.createContext()`. This lets you consume the nearest current value of that Context type using `this.context`. You can reference this in any of the lifecycle methods including the render function.

Context.Provider

```
<MyContext.Provider value={/* some value */}>
```

Every Context object comes with a Provider React component that allows consuming components to subscribe to context changes.

Accepts a `value` prop to be passed to consuming components that are descendants of this Provider. One Provider can be connected to many consumers. Providers can be nested to override values deeper within the tree.

All consumers that are descendants of a Provider will re-render whenever the Provider's `value` prop changes. The propagation from Provider to its descendant consumers (including `.contextType` and `useContext`) is not subject to the `shouldComponentUpdate` method, so the consumer is updated even when an ancestor component skips an update.

React.createContext

```
const MyContext = React.createContext(defaultValue);
```

Creates a Context object. When React renders a component that subscribes to this Context object it will read the current context value from the closest matching `Provider` above it in the tree.

The `defaultValue` argument is **only** used when a component does not have a matching Provider above it in the tree. This can be helpful for testing components in isolation without wrapping them. Note: passing `undefined` as a Provider value does not cause consuming components to use `defaultValue`.

Class.contextType

The contextType property on a class used to assign a Context object which is created by React.createContext(). It allows you to consume the closest current value of that Context type using this.context. We can reference this in any of the component life-cycle methods, including the render function.

Note: We can only subscribe to a single context using this API. If we want to use the experimental public class field's syntax, we can use a static class field to initialize the contextType.

Context.Consumer

It is the React component which subscribes to the context changes. It allows us to subscribe to the context within the function component. It requires the function as a component. A consumer is used to request data through the provider and manipulate the central data store when the provider allows it.

Syntax

```
<MyContext.Consumer>
    {value => /* render something which is based on the context value */}
</MyContext.Consumer>
```

The function component receives the current context value and then returns a React node. The value argument which passed to the function will be equal to the value prop of the closest Provider for this context in the component tree. If there is no Provider for this context, the value argument will be equal to the defaultValue which was passed to createContext().

Context.Provider

Every Context object has a Provider React component which allows consuming components to subscribe to context changes. It acts as a delivery service. When a consumer component asks for something, it finds it in the context and provides it to where it is needed.

Syntax

```
<MyContext.Provider value={/* some value */}>
```

It accepts the value prop and passes to consuming components which are descendants of this Provider. We can connect one Provider with many consumers. Context Providers can be nested to override values deeper within the component tree. All consumers that are descendants of a Provider always re-render whenever the Provider's value prop is changed. The changes are determined by comparing the old and new values using the same algorithm as **Object.is** algorithm.

React.createContext

It creates a context object. When React renders a component which subscribes to this context object, then it will read the current context value from the matching provider in the component tree.

Syntax

```
const MyContext = React.createContext(defaultValue);
```

When a component does not have a matching Provider in the component tree, it returns the defaultValue argument. It is very helpful for testing components isolation (separately) without wrapping them.

React Context API

The React Context API is a component structure, which allows us to share data across all levels of the application. The main aim of Context API is to solve the problem of prop drilling (also called "Threading"). The Context API in React are given below.

1. React.createContext
2. Context.provider
3. Context.Consumer
4. Class.contextType

When to use Context

Context is used to share data which can be considered "global" for React components tree and use that data where needed, such as the current authenticated user, theme, etc. For example, in the below code snippet, we manually thread through a "theme" prop to style the Button component.

```
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} />;
  }
}
```

In the above code, the Toolbar function component takes an extra "theme" prop and pass it to the ThemedButton. It can become inconvenient if every single button in the app needs to know the theme because it would be required to pass through all components. But using context, we can avoid passing props for every component through intermediate elements.

React Context

Context allows passing data through the component tree without passing props down manually at every level.

In React application, we passed data in a top-down approach via props. Sometimes it is inconvenient for certain types of props that are required by many components in the React application. Context provides a way to pass values between components without explicitly passing a prop through every level of the component tree.

How to use Context

There are two main steps to use the React context into the React application:

1. Setup a context provider and define the data which you want to store.
2. Use a context consumer whenever you need the data from the store

Context Consumer

A React component that subscribes to context changes. This lets you subscribe to a context within a function component.

It requires a function as a child. The function receives the current context value and returns a React node.

The *value* argument passed to the function will be equal to the *value* prop of the closest *Provider* for this context above in the tree.

If there is no *Provider* for this context above, the *value* argument will be equal to the *defaultValue* that was passed to createContext().

```
<MyContext.Consumer>
  {value => /* render something based on the context value */}
</MyContext.Consumer>
```

Context Provider

Every Context object comes with a *Provider* React component that allows consuming components to subscribe to context changes.

One Provider can be connected to many consumers. Providers can be nested to override values deeper within the tree.

Syntax:-

```
<MyContext.Provider value={/* some value */}>
```

A *value* prop to be passed to consuming components that are descendants of this *Provider*.

Ex:-

```
<MyContext.Provider value={this.state.name}>
```

Create Context

It creates a Context object.

When React renders a component that subscribes to this Context object it will read the current context value from the closest matching *Provider* above it in the tree.

Syntax: -

```
const MyContext = React.createContext(defaultValue);
```

defaultValue - It is only used when a component does not have a matching *Provider* above it in the tree.

Ex:-

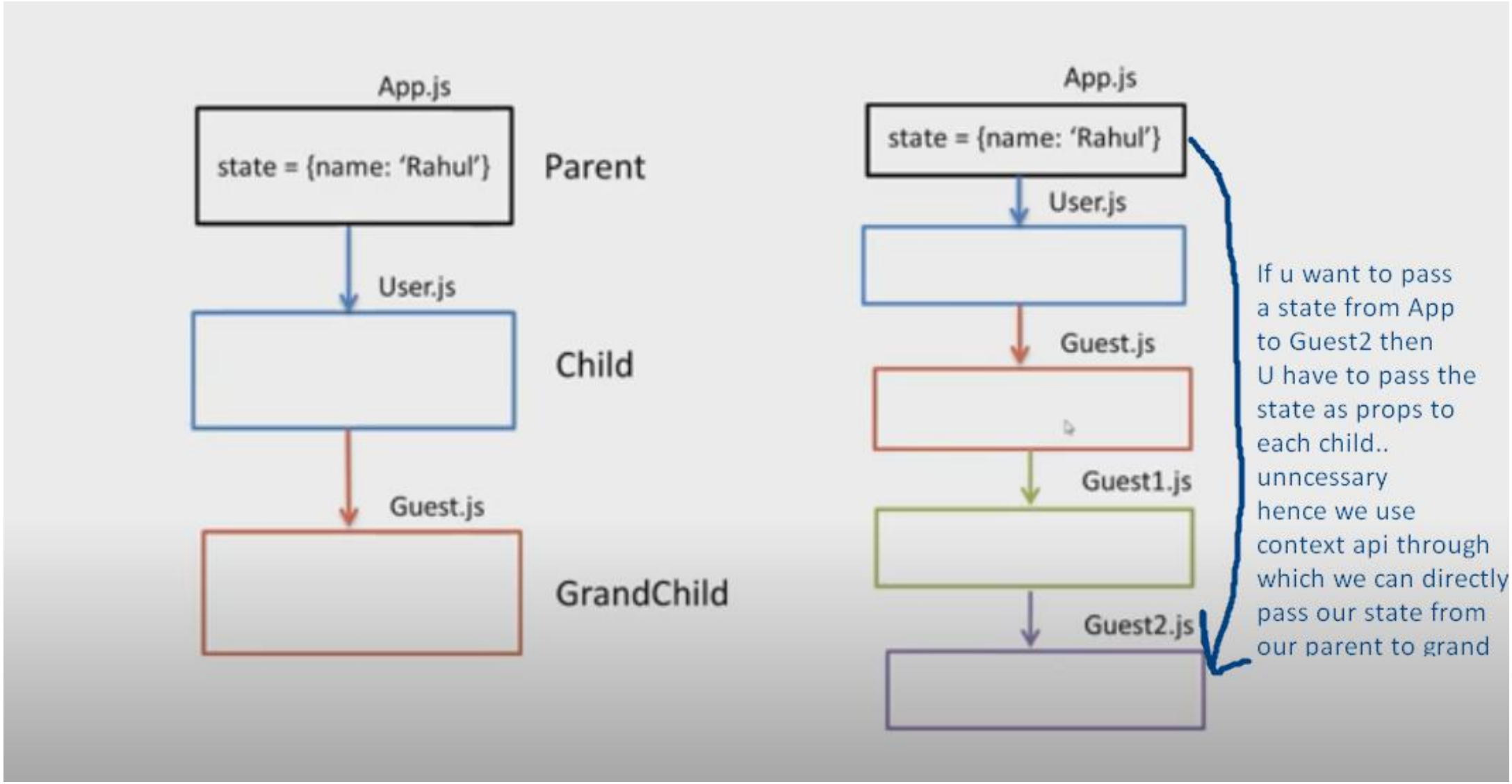
```
const MyContext = React.createContext(false);
const MyContext = React.createContext('white');
const MyContext = React.createContext({ user: 'Guest', });
```

Context

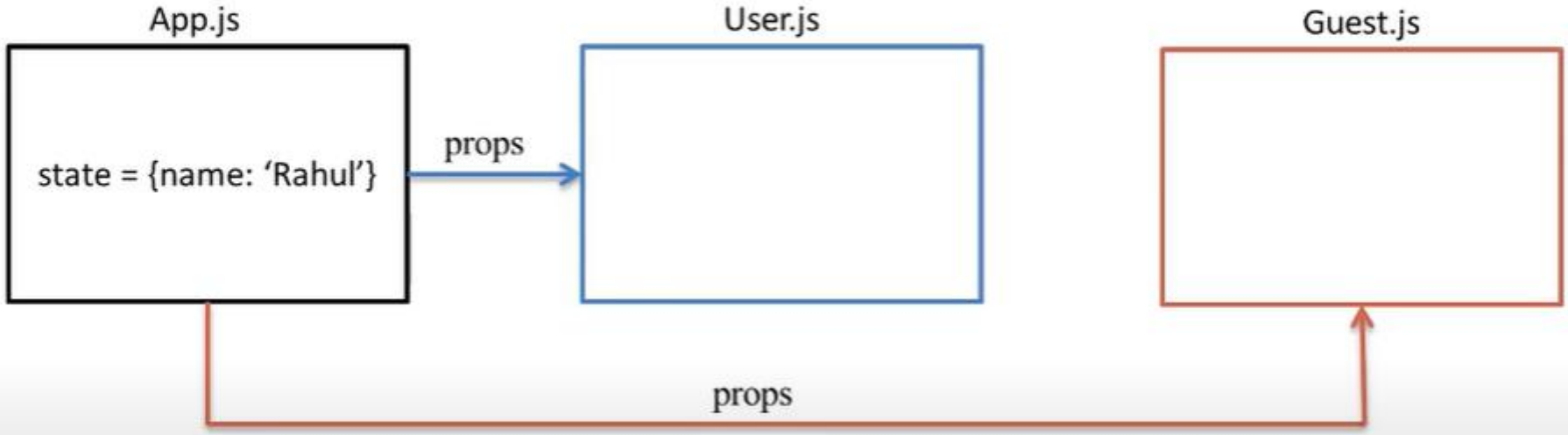
Context provides a way to pass data through the component tree without having to pass props down manually at every level.

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 that are required by many components within an application.

- Passing the initial state to *React.createContext*. This function then returns an object with a *Provider* and a *Consumer*.
- Using the *Provider* component at the top of the tree and making it accept a prop called *value*. This *value* can be anything!
- Using the *Consumer* component anywhere below the Provider in the component tree to get a subset of the state.


App.js
state = {name: 'Rahul'}
Parent
User.js
Child
Guest.js
GrandChild
App.js
state = {name: 'Rahul'}
User.js
Guest.js
Guest1.js
Guest2.js
If u want to pass
a state from App
to Guest2 then
U have to pass the
state as props to
each child..
unncessary
hence we use
context api through
which we can directly
pass our state from
our parent to grand

Lifting State up



InCase: if child component wants the same state as its parent component then we use the concept of lifting-State-U
as we know state are local to component that is they cannot be used apart form the component in which it made..
So what we do is, we send the state from parent component as a props to the child component and the child can us
that props to make it s own state which is same as it s parent component sta

React will assign the `current` property with the DOM element when the component mounts, and assign it back to `null` when it unmounts. `ref` updates happen before `componentDidMount` or `componentDidUpdate` lifecycle methods.

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // create a ref to store the textInput DOM element
    this.textInput = React.createRef();
    this.focusTextInput = this.focusTextInput.bind(this);
  }

  focusTextInput() {
    // Explicitly focus the text input using the raw DOM
API
    // Note: we're accessing "current" to get the DOM node
    this.textInput.current.focus();
  }

  render() {
    // tell React that we want to associate the <input> ref
    // with the `textInput` that we created in the
constructor
    return (
      <div>
        <input
          type="text"
          ref={this.textInput} />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

Accessing Refs

When a ref is passed to an element in `render`, a reference to the node becomes accessible at the `current` attribute of the ref.

```
const node = this.myRef.current;
```

The value of the ref differs depending on the type of the node:

- 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 mounted instance of the component as its `current`.
- **You may not use the `ref` attribute on function components** because they don't have instances.

Creating Refs

Refs are created using `React.createRef()` and attached to React elements via the `ref` attribute. Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}
```

Refs provide a way to access DOM nodes or React elements created in the render method.

In the typical React dataflow, props are the only way that parent components interact with their children. To modify a child, you re-render it with new props. However, there are a few cases where you need to imperatively modify a child outside of the typical dataflow. The child to be modified could be an instance of a React component, or it could be a DOM element. For both of these cases, React provides an escape hatch.

When to Use Refs

There are a few good use cases for refs:

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

Avoid using refs for anything that can be done declaratively.

For example, instead of exposing `open()` and `close()` methods on a `Dialog` component, pass an `isOpen` prop to it.

callback refs

React also supports another way to set refs called “callback refs”, which gives more fine-grain control over when refs are set and unset.

Instead of passing a *ref* attribute created by *createRef()*, you pass a function. The function receives the React component instance or HTML DOM element as its argument, which can be stored and accessed elsewhere.

The file input Tag

In HTML, an `<input type="file">` lets the user choose one or more files from their device storage to be uploaded to a server or manipulated by JavaScript via the File API.

```
<input type="file" />
```

In React, an `<input type="file" />` is always an uncontrolled component because its value can only be set by a user, and not programmatically.

You should use the File API to interact with the files. The following example shows how to create a ref to the DOM node to access file(s) in a submit handler:

Default Values

In the React rendering lifecycle, the `value` attribute on form elements will override the value in the DOM. With an uncontrolled component, you often want React to specify the initial value, but leave subsequent updates uncontrolled. To handle this case, you can specify a `defaultValue` attribute instead of `value`.

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Name:
        <input
          defaultValue="Bob"
          type="text"
          ref={this.input} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
}
```

Likewise, `<input type="checkbox">` and `<input type="radio">` support `defaultChecked`, and `<select>` and `<textarea>` supports `defaultValue`.

Accessing Refs

When a *ref* is passed to an element in render, a reference to the node becomes accessible at the *current* attribute of the *ref*.

```
const node = this.myRef.current;
```

React will assign the current property with the DOM element when the component mounts, and assign it back to null when it unmounts.

The value of the ref differs depending on the type of the node:

- 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 mounted instance of the component as its current.
- You may not use the *ref* attribute on function components because they don't have instances.

Creating Refs

Refs are created using *React.createRef()* and attached to React elements via the ***ref*** attribute.

Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

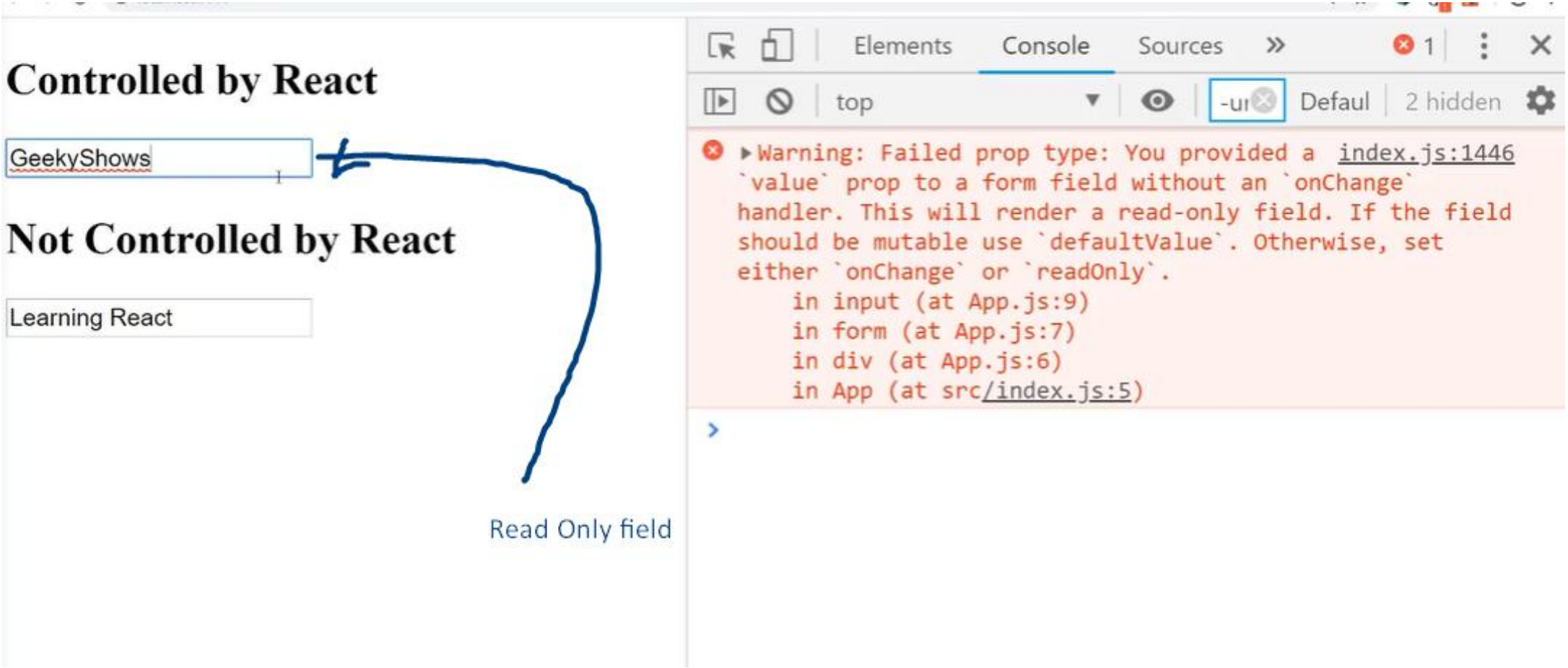
```
// Create a ref to store the DOM element
this.myRef = React.createRef();


render() {
    // Attaching created ref to react element
   return <div ref={this.myRef} />;
```

refs

Refs provide a way to access DOM nodes or React elements created in the render method.

When to Use Refs

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.


Controlled by React
GeekyShows
Not Controlled by React
Learning React
Read Only field
Elements
Console
Sources
1
top
Defaul
2 hidden
Warning: Failed prop type: You provided a index.js:1446
`value` prop to a form field without an `onChange`
handler. This will render a read-only field. If the field
should be mutable use `defaultValue`. Otherwise, set
either `onChange` or `readOnly`.
in input (at App.js:9)
in form (at App.js:7)
in div (at App.js:6)
in App (at src/index.js:5)

Uncontrolled Component

In a controlled component, form data is handled by a React component. The alternative is uncontrolled components, where form data is handled by the DOM itself.

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.

When Use Uncontrolled Component-

You do not need to write an event handler for every way your data can change and pipe all of the input state through a React component.

Converting a preexisting codebase to React, or integrating a React application with a non-React library.

Controlled Component

Form has the default HTML form behavior of browsing to a new page when the user submits the form. If you want this behavior in React, it just works. But in most cases, it's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered into the form. The standard way to achieve this is with a technique called "controlled components".

In a controlled component, form data is handled by a React component.

When Use Controlled Component-

You need to write an event handler for every way your data can change and pipe all of the input state through a React component.

Forms

HTML form elements work a little bit differently from other DOM elements in React, because form elements naturally keep some internal state.

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 (changeable) state is typically kept in the state property of components, and only updated with setState().

- Controlled Component
- Uncontrolled Component

Installation

The best way to consume React-Bootstrap is via the npm package which you can install with `npm` (or `yarn` if you prefer).

If you plan on customizing the Bootstrap Sass files, or don't want to use a CDN for the stylesheet, it may be helpful to install vanilla Bootstrap as well.

```
npm install react-bootstrap bootstrap
```

Importing Components

You should import individual components like: `react-bootstrap/Button` rather than the entire library. Doing so pulls in only the specific components that you use, which can significantly reduce the amount of code you end up sending to the client.

```
import Button from 'react-bootstrap/Button';

// or less ideally
import { Button } from 'react-bootstrap';
```

Browser globals

We provide `react-bootstrap.js` and `react-bootstrap.min.js` bundles with all components exported on the `window.ReactBootstrap` object. These bundles are available on unpkg, as well as in the npm package.

```
<script src="https://unpkg.com/react/umd/react.production.min.js" crossorigin></script>

<script
  src="https://unpkg.com/react-dom/umd/react-dom.production.min.js"
  crossorigin></script>

<script
  src="https://unpkg.com/react-bootstrap@next/dist/react-bootstrap.min.js"
  crossorigin></script>

<script>var Alert = ReactBootstrap.Alert;</script>
```

Examples

React-Bootstrap has started a repo with a few basic CodeSandbox examples. Click here to check them out.

Stylesheets

Because React-Bootstrap doesn't depend on a very precise version of Bootstrap, we don't ship with any included CSS. However, some stylesheet **is required** to use these components.

CSS

```
{/* The following line can be included in your src/index.js or App.js file*/}

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

How and which Bootstrap styles you include is up to you, but the simplest way is to include the latest styles from the CDN. A little more information about the benefits of using a CDN can be found here.

```
<link
  rel="stylesheet"
  href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css"
  integrity="sha384-9aIt2nRpC12Uk9gS9baDl411NQApFmC26EwAOH8WgZl5MYYxFfc+NcPb1dKGj7Sk"
  crossorigin="anonymous"
/>
```

Sass

In case you are using **Sass** the simplest way is to include the Bootstrap's source Sass files in your main Sass file and then require it on your `src/index.js` or `App.js` file. This applies to a typical `create-react-app` application in other use cases you might have to setup the bundler of your choice to compile Sass/SCSS stylesheets to CSS.

```
/* The following line can be included in a src/App.scss */

@import "~bootstrap/scss/bootstrap";

/* The following line can be included in your src/index.js or App.js file */

import './App.scss';
```

Customize Bootstrap

If you wish to customize the Bootstrap theme or any Bootstrap variables you can create a custom Sass file:

```
/* The following block can be included in a custom.scss */

/* make the customizations */
$theme-colors: (
    "info": tomato,
    "danger": teal
);

/* import bootstrap to set changes */
@import "~bootstrap/scss/bootstrap";
```



```
/* The following line can be included in a src/App.scss */

@import "custom";
```

React Bootstrap Installation

Let us create a new React app using the **create-react-app** command as follows.

```
$ npx create-react-app react-bootstrap-app
```

After creating the React app, the best way to install Bootstrap is via the npm package. To install Bootstrap, navigate to the React app folder, and run the following command.

```
$ npm install react-bootstrap bootstrap --save
```

Importing Bootstrap

Now, open the **src/index.js** file and add the following code to import the Bootstrap file.

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

We can also import individual components **like import { SplitButton, Dropdown } from 'react-bootstrap';** instead of the entire library. It provides the specific components which we need to use, and can significantly reduce the amount of code.

In the React app, create a new file named **ThemeSwitcher.js** in the **src** directory and put the following code.

```
import React, { Component } from 'react';
import { SplitButton, Dropdown } from 'react-bootstrap';

class ThemeSwitcher extends Component {

 state = { theme: null }

 chooseTheme = (theme, evt) => {
  evt.preventDefault();
  if (theme.toLowerCase() === 'reset') { theme = null }
  this.setState({ theme });
 }

 render() {
  const { theme } = this.state;
  const themeClass = theme ? theme.toLowerCase() : 'default';

  const parentContainerStyles = {
   position: 'absolute',
   height: '100%',
   width: '100%',
   display: 'table'
  };

  const subContainerStyles = {
   position: 'relative',
   height: '100%',
   width: '100%',
   display: 'table-cell',
  };

  return (
   <div style={parentContainerStyles}>
    <div style={subContainerStyles}>

     <span className={`h1 center-block text-
center text-${theme ? themeClass : 'muted'}`} style={{ marginBottom: 25 }}>{theme || 'Default'}</span>

     <div className="center-block text-center">
      <SplitButton bsSize="large" bsStyle={themeClass} title={`${theme || 'Default Block'} Theme`}>
       <Dropdown.Item eventKey="Primary Block" onSelect={this.chooseTheme}>Primary Theme</Dropdown.Item>
       <Dropdown.Item eventKey="Danger Block" onSelect={this.chooseTheme}>Danger Theme</Dropdown.Item>
       <Dropdown.Item eventKey="Success Block" onSelect={this.chooseTheme}>Success Theme</Dropdown.Item>
       <Dropdown.Item divider />
       <Dropdown.Item eventKey="Reset Block" onSelect={this.chooseTheme}>Default Theme</Dropdown.Item>
      </SplitButton>
     </div>
    </div>
   </div>
  );
 }
}
export default ThemeSwitcher;
```

Now, update the **src/index.js** file with the following snippet.

Index.js

```
import 'bootstrap/dist/css/bootstrap.min.css';
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';
import './index.css';
import ThemeSwitcher from './ThemeSwitcher';

ReactDOM.render(<ThemeSwitcher />, document.getElementById('root'));
```

Using reactstrap

Let us create a new React app using the create-react-app command as follows.

```
$ npx create-react-app reactstrap-app
```

Next, install the **reactstrap** via the npm package. To install reactstrap, navigate to the React app folder, and run the following command.

```
$ npm install bootstrap reactstrap --save
```

Importing Bootstrap

Now, open the **src/index.js** file and add the following code to import the Bootstrap file.

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

We can also import individual components **like import { Button, Dropdown } from 'reactstrap';** instead of the entire library. It provides the specific components which we need to use, and can significantly reduce the amount of code.

In the React app, create a new file named **ThemeSwitcher.js** in the **src** directory and put the following code.

```
import React, { Component } from 'react';
import { Button, ButtonDropdown, DropdownToggle, DropdownMenu, DropdownItem } from 'reactstrap';

class ThemeSwitcher extends Component {

 state = { theme: null, dropdownOpen: false }

 toggleDropdown = () => {
  this.setState({ dropdownOpen: !this.state.dropdownOpen });
 }

 resetTheme = evt => {
  evt.preventDefault();
  this.setState({ theme: null });
 }

 chooseTheme = (theme, evt) => {
  evt.preventDefault();
  this.setState({ theme });
 }
 render() {
  const { theme, dropdownOpen } = this.state;
  const themeClass = theme ? theme.toLowerCase() : 'secondary';

  return (
   <div className="d-flex flex-wrap justify-content-center align-items-center">

    <span className={`h1 mb-4 w-100 text-center text-${themeClass}`}>{theme || 'Default'}</span>

    <ButtonDropdown isOpen={dropdownOpen} toggle={this.toggleDropdown}>
     <Button id="caret" color={themeClass}>{theme || 'Custom'} Theme</Button>
     <DropdownToggle caret size="lg" color={themeClass} />
     <DropdownMenu>
      <DropdownItem onClick={e => this.chooseTheme('Primary', e)}>Primary Theme</DropdownItem>
      <DropdownItem onClick={e => this.chooseTheme('Danger', e)}>Danger Theme</DropdownItem>
      <DropdownItem onClick={e => this.chooseTheme('Success', e)}>Success Theme</DropdownItem>
      <DropdownItem divider />
      <DropdownItem onClick={this.resetTheme}>Default Theme</DropdownItem>
     </DropdownMenu>
    </ButtonDropdown>

   </div>
  );
 }
}
export default ThemeSwitcher;
```

Now, update the **src/index.js** file with the following snippet.

Index.js

```
import 'bootstrap/dist/css/bootstrap.min.css';
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';
import './index.css';
import ThemeSwitcher from './ThemeSwitcher';

ReactDOM.render(<ThemeSwitcher />, document.getElementById('root'));
```

React Bootstrap Package

The React Bootstrap package is the most popular way to add bootstrap in the React application. There are many Bootstrap packages built by the community, which aim to rebuild Bootstrap components as React components. The **two** most popular Bootstrap packages are:

1. **react-bootstrap:** It is a complete re-implementation of the Bootstrap components as React components. It does not need any dependencies like bootstrap.js or jQuery. If the React setup and React-Bootstrap installed, we have everything which we need.
2. **reactstrap:** It is a library which contains React Bootstrap 4 components that favor composition and control. It does not depend on jQuery or Bootstrap JavaScript. However, react-popper is needed for advanced positioning of content such as Tooltips, Popovers, and auto-flipping Dropdowns.

React Bootstrap Installation

Let us create a new React app using the **create-react-app** command as follows.

```
$ npx create-react-app react-bootstrap-app
```

After creating the React app, the best way to install Bootstrap is via the npm package. To install Bootstrap, navigate to the React app folder, and run the following command.

```
$ npm install react-bootstrap bootstrap --save
```

Bootstrap as Dependency

If we are using a build tool or a module bundler such as Webpack, then importing Bootstrap as dependency is the preferred option for adding Bootstrap to the React application. We can install Bootstrap as a dependency for the React app. To install the Bootstrap, run the following commands in the terminal window.

```
$ npm install bootstrap --save
```

Once Bootstrap is installed, we can import it in the React application entry file. If the React project created using the **create-react-app** tool, open the **src/index.js** file, and add the following code:

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

Now, we can use the CSS classes and utilities in the React application. Also, if we want to use the JavaScript components, we need to install the **jquery** and **popper.js** packages from **npm**. To install the following packages, run the following command in the terminal window.

```
$ npm install jquery popper.js
```

Next, go to the **src/index.js** file and add the following imports.

```
import $ from 'jquery';
import Popper from 'popper.js';
import 'bootstrap/dist/js/bootstrap.bundle.min';
```

Now, we can use Bootstrap JavaScript Components in the React application.

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">
```

If there is a need to use Bootstrap components which depend on JavaScript/jQuery in the React application, we need to include **jQuery**, **Popper.js**, and **Bootstrap.js** in the document. Add the following imports in the **<script>** tags near the end of the closing **</body>** tag of the **index.html** file.

```
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js" integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1" crossorigin="anonymous"></script>

<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js" integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM" crossorigin="anonymous"></script>
```

In the above snippet, we have used jQuery's slim version, although we can also use the full version as well. Now, Bootstrap is successfully added in the React application, and we can use all the CSS utilities and UI components available from Bootstrap in the React application.

Using the Bootstrap CDN

It is the easiest way of adding Bootstrap to the React app. There is no need to install or download Bootstrap. We can simply put an **<link>** into the **<head>** section of the **index.html** file of the React app as shown in the following snippet.

React Bootstrap

← prev next →

Single-page applications gaining popularity over the last few years, so many front-end frameworks have introduced such as Angular, React, Vue.js, Ember, etc. As a result, jQuery is not a necessary requirement for building web apps. Today, React has the most used JavaScript framework for building web applications, and Bootstrap become the most popular CSS framework. So, it is necessary to learn various ways in which Bootstrap can be used in React apps, which is the main aim of this section.

Adding Bootstrap for React

We can add Bootstrap to the React app in several ways. The **three** most common ways are given below:

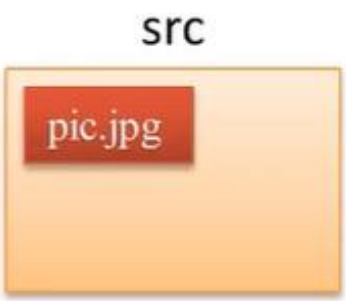
1. Using the Bootstrap CDN
2. Bootstrap as Dependency
3. React Bootstrap Package

Inside src Folder

How to Use

App.js

```
import pic from './pic.jpg';
<img src={pic} alt="mypic" />
```



This ensures that when the project is built, Webpack will correctly move the images into the build folder, and provide us with correct paths.

Inside src Folder

With Webpack, using static assets like images and fonts works similarly to CSS.

You can import a file right in a JavaScript module. This tells Webpack to include that file in the bundle. Unlike CSS imports, importing a file gives you a string value. This value is the final path you can reference in your code, e.g. as the src attribute of an image or the href of a link to a PDF.

- Scripts and stylesheets get minified and bundled together to avoid extra network requests.
- Missing files cause compilation errors instead of 404 errors for your users.
- Result filenames include content hashes so you don't need to worry about browsers caching their old versions.

Inside public Folder

How to use

Public Folder -> index.html

```
<img src="%PUBLIC_URL%/pic.jpg" alt="mypic" />
<img src="%PUBLIC_URL%/image/pic.jpg" alt="mypic" />
```

App.js

```
<img src={process.env.PUBLIC_URL + "/pic.jpg"}  />
<img src={process.env.PUBLIC_URL + "/image/pic.jpg"} />
```

Inside public Folder

When use Public Folder

- You need a file with a specific name in the build output, such as manifest.webmanifest.
- You have thousands of images and need to dynamically reference their paths.
- You want to include a small script like pace.js outside of the bundled code.
- Some library may be incompatible with Webpack and you have no other option but to include it as a <script> tags

Inside public Folder

If you put a file into the public folder, it will not be processed by Webpack. Instead it will be copied into the build folder untouched.

To reference assets in the public folder, you need to use a special variable called PUBLIC_URL. Only files inside the public folder will be accessible by %PUBLIC_URL% prefix.

Normally we recommend importing stylesheets, images, and fonts from JavaScript.

<link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">

- None of the files in public folder get post-processed or minified.
- Missing files will not be called at compilation time, and will cause 404 errors for your users.
- Result filenames won't include content hashes so you'll need to add query arguments or rename them every time they change.

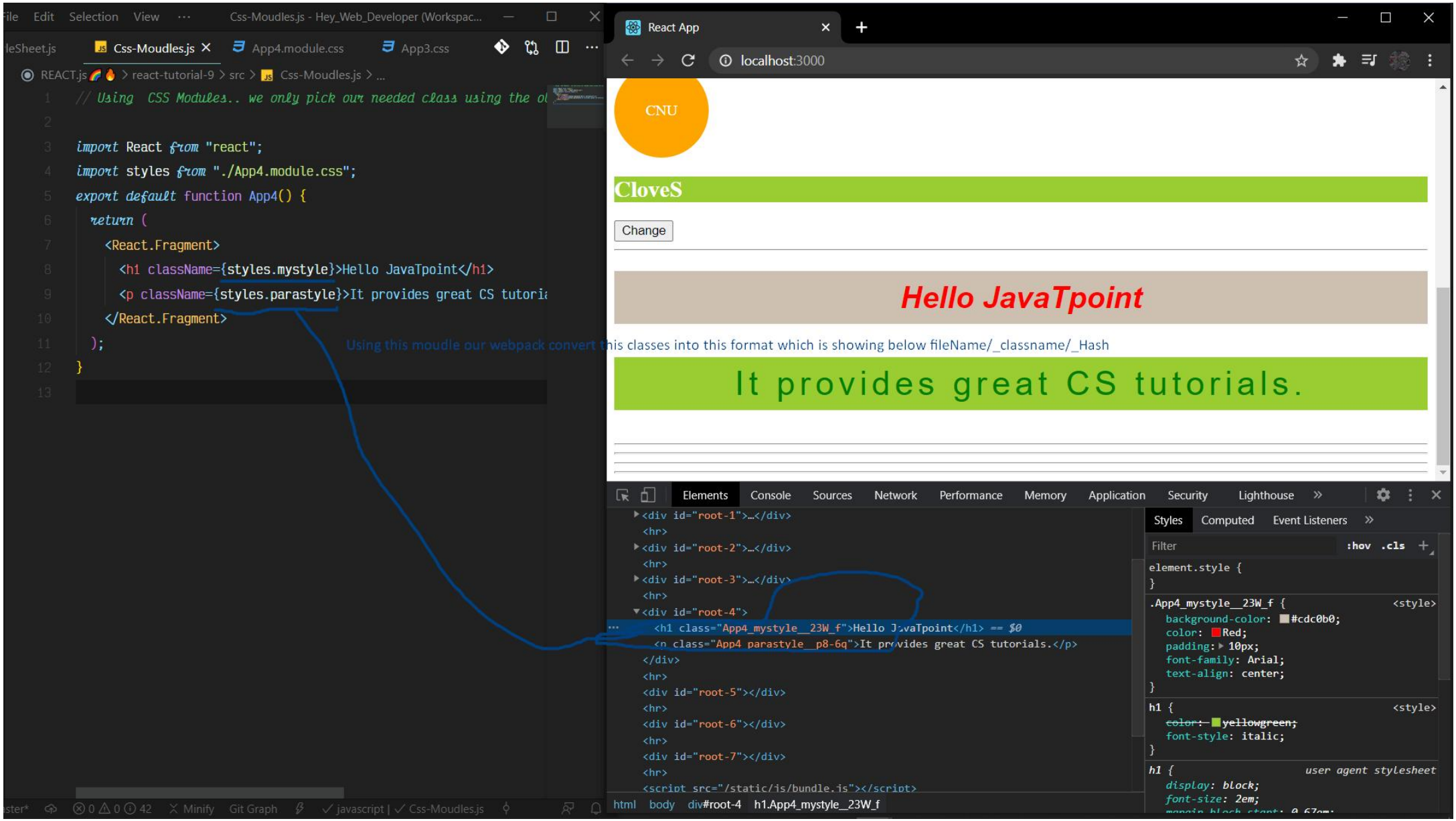
Images/Assets in React JS

- Inside public Folder
- Inside src Folder

CSS in JS

"CSS-in-JS" refers to a pattern where CSS is composed using JavaScript instead of defined in external files. This functionality is not a part of React, but provided by third-party libraries.

- Glamorous
- Styled Component
- Radium
- Emotion


File Edit Selection View ···
Css-Moudles.js - Hey_Web_Developer (Workspac...
leSheet.js
Css-Moudles.js
App4.module.css
App3.css
REACT.js
react-tutorial-9 > src > Css-Moudles.js > ...
// Using CSS Modules.. we only pick our needed class using the ol
import React from "react";
import styles from "./App4.module.css";
export default function App4() {
return (
<React.Fragment>
<h1 className={styles.mystyle}>Hello JavaTpoint</h1>
It provides great CS tutoria
</React.Fragment>
);
}
Using this moudle our webpack convert this classes into this format which is showing below fileName/_classname/_Hash
React App
localhost:3000
CNU
CloveS
Change
Hello JavaTpoint
It provides great CS tutorials.
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
…
<hr>
…
<hr>
…
<hr>

<h1 class="App4_mystyle__23W_f">Hello JavaTpoint</h1> == $0
It provides great CS tutorials.

<hr>

<hr>

<hr>

<hr>
<script src="/static/js/bundle.js"></script>
html body div#root-4 h1.App4_mystyle__23W_f
Styles
Computed
Event Listeners
Filter
:hov .cls
element.style {
}
.App4_mystyle__23W_f {
background-color: #cdc0b0;
color: Red;
padding: 10px;
font-family: Arial;
text-align: center;
}
h1 {
color: yellowgreen;
font-style: italic;
}
h1 {
display: block;
font-size: 2em;
user agent stylesheet

Import CSS Module and Regular CSS

CSS File Names

jai.module.css

veeru.css

Import

```
import styles from "./jai.module.css"
import "./veeru.css"
```

CSS Module

CSS Modules let you use the same CSS class name in different files without worrying about naming clashes.

CSS files in which all class names and animation names are scoped locally by default.

CSS Modules allows the scoping of CSS by automatically creating a unique classname of the format [filename]_[classname]__[hash]

Syntax:-

[name].module.css

This feature is available with react-scripts@2.0.0 and higher.

Ex:-

FileName:

App.module.css

```
import styles from "./App.module.css"
<h1 className={styles.txt}>Hello</h1>;
```

External Stylesheet

App.css

```
.txt {
  color: blue;
}
```

App.js

```
import './App.css';                    // This tells Webpack that App.js uses these styles.
<h1 className="txt">Hello App</h1>
```

Note:-

- Use className not class e.g. className="txt"
- Pass a string as the className prop.
- It is common for CSS classes to depend on the component props or state.
- In production, all CSS files will be concatenated into a single minified .css file in the build output.

Using JavaScript Object

The inline styling also allows us to create an object with styling information and refer it in the style attribute.

Example

App.js

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
 render() {
  const mystyle = {
   color: "Green",
   backgroundColor: "lightBlue",
   padding: "10px",
   fontFamily: "Arial"
  };
  return (
   <div>
   <h1 style={mystyle}>Hello JavaTpoint</h1>
   <p>Here, you can find all CS tutorials.</p>
   </div>
  );
 }
}
export default App;
```

camelCase Property Name

If the properties have two names, like **background-color**, it must be written in camel case syntax.

Example

App.js

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
  render() {
    return (
      <div>
      <h1 style={{color: "Red"}}>Hello JavaTpoint!</h1>
      <p style={{backgroundColor: "lightgreen"}}>Here, you can find all CS tutorials.</p>
      </div>
    );
  }
}
export default App;
```

1. Inline Styling

The inline styles are specified with a JavaScript object in camelCase version of the style name. Its value is the style?s value, which we usually take in a string.

Example

App.js

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
 render() {
   return (
     <div>
     <h1 style={{color: "Green"}}>Hello JavaTpoint!</h1>
     <p>Here, you can find all CS tutorials.</p>
     </div>
   );
 }
}
export default App;
```

React CSS

← prev next →

CSS in React is used to style the React App or Component. The **style** attribute is the most used attribute for styling in React applications, which adds dynamically-computed styles at render time. It accepts a JavaScript object in **camelCased** properties rather than a CSS string. There are many ways available to add styling to your React App or Component with CSS. Here, we are going to discuss mainly **four** ways to style React Components, which are given below:

1. Inline Styling
2. CSS Stylesheet
3. CSS Module
4. Styled Components

Inline Stylesheet

style is most often used in React applications to add dynamically-computed styles at render time.

The *style* attribute accepts a JavaScript object with camelCased properties rather than a CSS string. This is consistent with the DOM style JavaScript property, is more efficient, and prevents XSS security holes.

CSS classes are generally better for performance than inline styles.

styles are not autoprefixed. Vendor prefixes other than ms should begin with a capital letter e.g. WebkitTransition has an uppercase "W"

Ex:-

```
const btnStyle = {
  color: 'blue',
  backgroundColor: 'orange',
};
<button style={btnStyle}>Button</button>;
```

```
btnStyle.color = 'orange'
btnStyle.backgroundColor = 'blue'
```

Example

```
import React from 'react';
import ReactDOM from 'react-dom';
function MenuBlog(props) {
  const titlebar = (
    <ol>
      {props.data.map((show) =>
        <li key={show.id}>
          {show.title}
        </li>
      )}
    </ol>
  );
  const content = props.data.map((show) =>
    <div key={show.id}>
      <h3>{show.title}: {show.content}</h3>
    </div>
  );
  return (
    <div>
      {titlebar}
      <hr />
      {content}
    </div>
  );
}
const data = [
  {id: 1, title: 'First', content: 'Welcome to JavaTpoint!!'},
  {id: 2, title: 'Second', content: 'It is the best ReactJS Tutorial!!'},
  {id: 3, title: 'Third', content: 'Here, you can learn all the ReactJS topics!!'}
];
ReactDOM.render(
  <MenuBlog data={data} />,
  document.getElementById('app')
);
export default App;
```

Uniqueness of Keys among Siblings

We had discussed that keys assignment in arrays must be unique among their **siblings**. However, it doesn't mean that the keys should be **globally** unique. We can use the same set of keys in producing two different arrays. It can be understood in the below example.

Example: Correct Key usage

To correct the above example, we should have to assign key to the map() iterator.

```
import React from 'react';
import ReactDOM from 'react-dom';

function ListItem(props) {
  const item = props.item;
  return (
    // No need to specify the key here.
    <li> {item} </li>
  );
}
function NameList(props) {
  const myLists = props.myLists;
  const listItems = myLists.map((strLists) =>
    // The key should have been specified here.
    <ListItem key={myLists.toString()} item={strLists} />
  );
  return (
    <div>
        <h2>Correct Key Usage Example</h2>
            <ol>{listItems}</ol>
    </div>
  );
}
const myLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];
ReactDOM.render(
  <NameList myLists={myLists}/>,
  document.getElementById('app')
);
export default App;
```

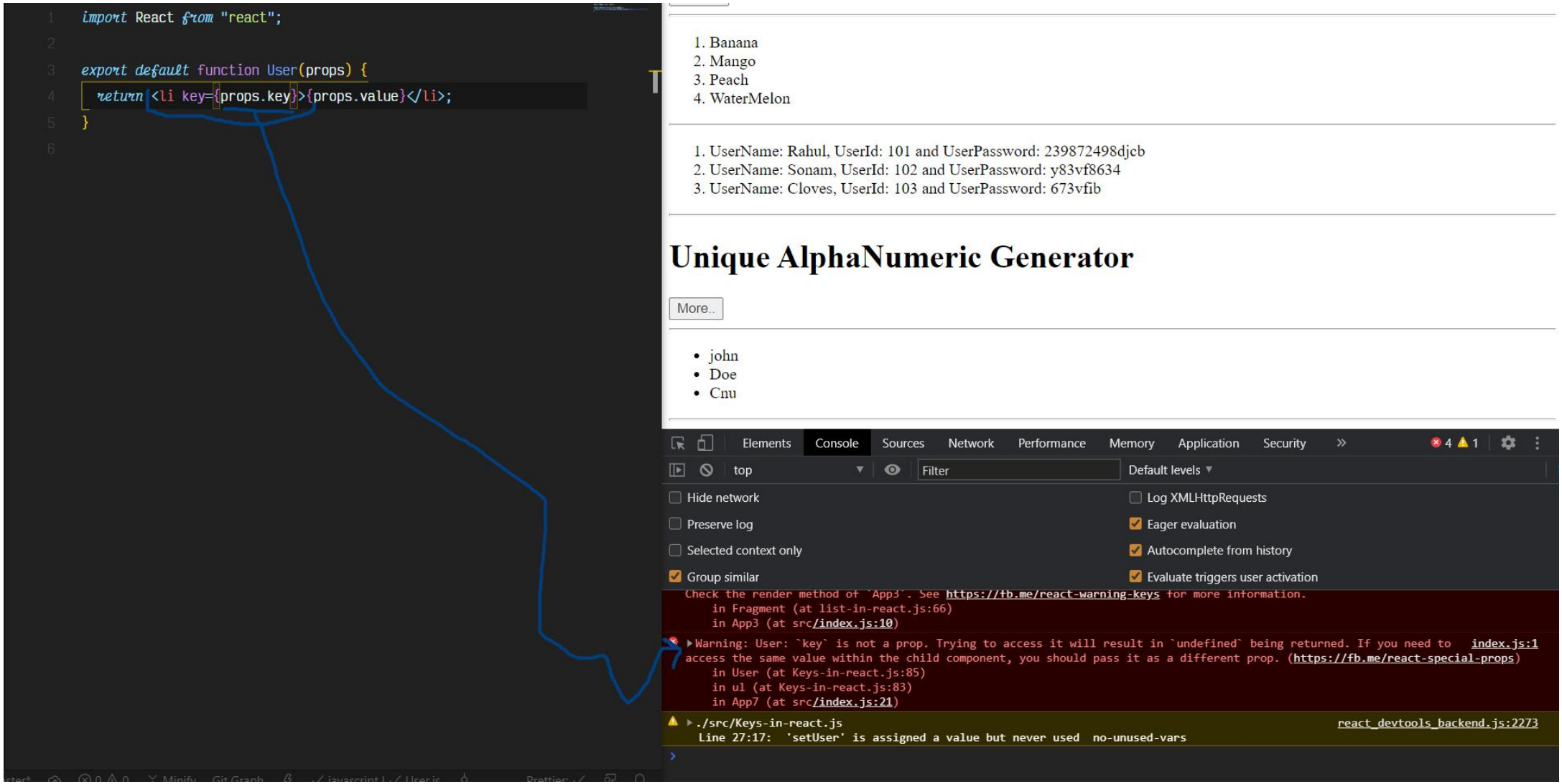
Example: Incorrect Key usage

```
import React from 'react';
import ReactDOM from 'react-dom';

function ListItem(props) {
  const item = props.item;
  return (
    // Wrong! No need to specify the key here.
    <li key={item.toString()}>
      {item}
    </li>
  );
}
function NameList(props) {
  const myLists = props.myLists;
  const listItems = myLists.map((strLists) =>
    // The key should have been specified here.
    <ListItem item={strLists} />
  );
  return (
    <div>
        <h2>Incorrect Key Usage Example</h2>
            <ol>{listItems}</ol>
    </div>
  );
}
const myLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];
ReactDOM.render(
  <NameList myLists={myLists}/>,
  document.getElementById('app')
);
export default App;
```

In the given example, the list is rendered successfully. But it is not a good practice that we had not assigned a key to the map() iterator.

Using Keys with component

Consider you have created a separate component for **ListItem** and extracting ListItem from that component. In this case, you should have to assign keys on the **<ListItem />** elements in the array, not to the **<li>** elements in the ListItem itself. To avoid mistakes, you have to keep in mind that keys only make sense in the context of the surrounding array. So, anything you are returning from map() function is recommended to be assigned a key.


import React from "react";
export default function User(props) {
return <li key={props.key}>{props.value};
}
1. Banana
2. Mango
3. Peach
4. WaterMelon
1. UserName: Rahul, UserId: 101 and UserPassword: 239872498djcb
2. UserName: Sonam, UserId: 102 and UserPassword: y83vf8634
3. UserName: Cloves, UserId: 103 and UserPassword: 673vfib
Unique AlphaNumeric Generator
More..
john
Doe
Cnu
Elements
Console
Sources
Network
Performance
Memory
Application
Security
top
Filter
Default levels
Hide network
Preserve log
Selected context only
Group similar
Log XMLHttpRequests
Eager evaluation
Autocomplete from history
Evaluate triggers user activation
Check the render method of `App3`. See https://fb.me/react-warning-keys for more information.
in Fragment (at list-in-react.js:66)
in App3 (at src/index.js:10)
Warning: User: `key` is not a prop. Trying to access it will result in `undefined` being returned. If you need to access the same value within the child component, you should pass it as a different prop. (https://fb.me/react-special-props)
in User (at Keys-in-react.js:85)
in ul (at Keys-in-react.js:83)
in App7 (at src/index.js:21)
index.js:1
./src/Keys-in-react.js
Line 27:17: 'setUser' is assigned a value but never used no-unused-vars
react_devtools_backend.js:2273

Here's something that would return only unique alphanumerics

```
unction alphanumeric_unique() {
   return Math.random().toString(36).split('').filter( function(value, index, self) {
       return self.indexOf(value) === index;
   }).join('').substr(2,8);
```

FIDDLE

Splitting the string into an array of characters, then using `Array.filter()` to filter out any characters that are already in the array to get only one instance of each character, and then finally joining the characters back to a string, and running `substr(2, 8)` to get the same length string as in the question, where it starts at the second character and gets a total of eight characters.

```
▶Warning: Each child in a list should have a unique "key" prop.                index.js:1

Check the render method of `App`. See https://fb.me/react-warning-keys for more information.
    in li (at list-in-react.js:9)
    in App (at src/index.js:6)
▶Warning: Each child in a list should have a unique "key" prop.                index.js:1

Check the render method of `App2`. See https://fb.me/react-warning-keys for more information.
    in li (at list-in-react.js:39)
    in App2 (at src/index.js:9)
▶Warning: Each child in a list should have a unique "key" prop.                index.js:1

Check the render method of `App3`. See https://fb.me/react-warning-keys for more information.
    in Fragment (at list-in-react.js:66)
    in App3 (at src/index.js:10)
```

Keys

```
state = {
  users: [
   { id: 101, name: "Rahul"},
   { id: 102, name: "Sonam"},
   { id: 103, name: "Rahul"}
  ],
};
const newUsers = this.state.users.map(user => {
return (
     <h1>ID: {user.id} Name: {user.name} </h1>
   );
  });
```

```
state = {
  users: [
   { id: 101, name: "Rahul"},
   { id: 102, name: "Sonam"},
   { id: 103, name: "Rahul"}
  ],
};
const newUsers = this.state.users.map(user => {
return (
     <h1 key={user.id}>ID: {user.id} Name: {user.name}
</h1>
   );
  });
```

Note:-

- A good rule of thumb is that elements inside the map() call need keys.
- Key should be specified inside the array.

Keys

```
// Declaration and Initialization of Array
  const arr = [10, 20, 30, 40];

  // Using Array Map Method
  const newArr = arr.map(num => {
     return <li key={num}>{num * 2}</li>;
  });
```

```
// Declaration and Initialization of Array
  const arr = [10, 20, 30, 40];

  // Using Array Map Method
  const newArr = arr.map((num, i) => {
     return <li key={i}>{num * 2}</li>;
  });
```

*Not recommended

Note:-

- A good rule of thumb is that elements inside the map() call need keys.
- Key should be specified inside the array.

React Keys

A key is a unique identifier. In React, it is used to identify which items have changed, updated, or deleted from the Lists. It is useful when we dynamically created components or when the users alter the lists. It also helps to determine which components in a collection needs to be re-rendered instead of re-rendering the entire set of components every time.

Keys should be given inside the array to give the elements a stable identity. The best way to pick a key as a string that uniquely identifies the items in the list. It can be understood with the below example.

Example

```
const stringLists = [ 'Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa' ];

const updatedLists = stringLists.map((strList)=>{
   <li key={strList.id}> {strList} </li>;
});
```

Keys

- A "key" is a special string attribute you need to include when creating lists of elements.
- Keys help React identify which items have changed, are added, or are removed.
- Keys should be given to the elements inside the array to give the elements a stable identity.
- The best way to pick a key is to use a string that uniquely identifies a list item among its siblings.
- Most often you would use IDs from your data as keys.
- Keys used within arrays should be unique among their siblings. However they don't need to be globally unique. We can use the same keys when we produce two different arrays.
- Keys serve as a hint to React but they don't get passed to your components.
- If you need the same value in your component, pass it explicitly as a prop with a different name.

Objects

```
const Person = () => {
  const [person, setPerson] = useState({
    firstName: '',
    lastName: ''
  });

  const handleChange = (e) => {
    setPerson({
      ...person,
      [e.target.name]: e.target.value
    });
  };

  const handleSubmit = (e) => {
    e.preventDefault()
    // Form submission logic here.
  }

  return (
    <form>
      <label htmlFor='first'>
        First Name:
        <input
          id='first'
          name='firstName'
          type='text'
          value={person.firstName}
          onChange={handleChange}
        />
      </label>
      <label htmlFor='last'>
        Last Name:
        <input
          id='last'
          name='lastName'
          type='text'
          value={person.lastName}
          onChange={handleChange}
        />
      </label>
      <button type='submit' onClick={handleSubmit}>Submit</button>
    </form>
  );
};
```

In the above example, the `handleChange` function calls `setPerson` and passes in the person object from state using the spread operator with `...person`. Without passing in the existing person object stored in state, the entire object would be overwritten anytime one of the input values changed.

Here is the classic and simple counter component example. We want to increment or decrement a number stored in state and display that number to the user or reset that number back to 0.

```
import React, { useState } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0)

  const increment = () => setCount(count + 1)
  const decrement = () => setCount(count - 1)
  const reset = () => setCount(0)

  return (
    <div className='counter'>
      <p className='count'>{count}</p>
      <div className='controls'>
        <button onClick={increment}>Increment</button>
        <button onClick={decrement}>Decrement</button>
        <button onClick={reset}>Reset</button>
      </div>
    </div>
  )
}

export default Counter
```

Updating Primitive Types

Updating state variables with `useState` always replaces the previous state. This means that updating primitive types (strings, booleans, numbers) is simple because their values are replaced rather than mutated.

A Few Examples of State with Various Data Types

Each piece of state has its own call to `useState` and its own variable and function for setting/updating it.

```
const [count, setCount] = useState(0)
const [color, setColor] = useState('#526b2d')
const [isHidden, setIsHidden] = useState(true)
const [products, setProducts] = useState([])
const [user, setUser] = useState({
    username: '',
    avatar: '',
    email: '',
})
```

Create and Initialize State

When called, `useState` returns an array of two items. The first being our state value and the second being a function for setting or updating that value. The `useState` hook takes a single argument, the initial value for the associated piece of state, which can be of any Javascript data type.

We assign these two returned values to variables using array destructuring.

```
import React, { useState } from 'react';

const Component = () => {
    const [value, setValue] = useState(initial value)
    ...
```

Since array elements have no names, we can name these two variables whatever we want. The general convention for declaring the name of your updater function is to begin with set and end with the name of your state variable, so `[value, setValue]`. The initial state argument passed in will be the value assigned to the state variable on the first render.

Updating Arrays and Objects

When updating arrays or objects in state with `useState`, you must remember to pass the entire object or array to the updater function as the state is replaced, NOT merged as with the `setState` method found in class-based components.

Arrays

```
const [items, setItems] = useState([])

// Completely replaces whatever was stored in the items array
setItems([{item1}, {item2}])

// Don't use JS array methods such as pop, push, shift, unshift
// as these will not tell React to trigger a re-render.
items.push({item3})

// Instead, make a copy of the array then add your new item onto the end
setItems([...items, {item3}])

// To update an item in the array use .map.
// Assumes each array item is an object with an id.
setItems(
  items.map((item, index) => {
    item.id === id ? newItem : item
  })
)
```

React Lists

Lists are used to display data in an ordered format and mainly used to display menus on websites. In React, Lists can be created in a similar way as we create lists in JavaScript. Let us see how we transform Lists in regular JavaScript.

The map() function is used for traversing the lists. In the below example, the map() function takes an array of numbers and multiply their values with 5. We assign the new array returned by map() to the variable multiplyNums and log it.

Example

```
var numbers = [1, 2, 3, 4, 5];
const multiplyNums = numbers.map((number)=>{
   return (number * 5);
});
console.log(multiplyNums);
```

Iteration using map ( ) Method

The map( ) method creates a new array with the results of calling a provided function on every element in the calling array.

map calls a provided callback function once for each element in an array, in order, and returns a new array from the results.

Syntax:- map(callback(currentValue, index, array), thisArg);

Ex:- map((num, index) => {return num})

```
// Declaration and Initialization of Array
   const arr = [10, 20, 30, 40];

   // Using Array Map Method
   const newArr = arr.map(num => {
      return <li>{num * 2}</li>;
   });
```

Lists

You can build collections of elements and include them in JSX using curly braces {}.

```
const arr = [10, 20, 30, 40];
```

```
state = {
        users: [
                { id: 101, name: "Rahul", password: "3423ssdf" },
                { id: 102, name: "Sonam", password: "654yuei" },
                { id: 103, name: "Rahul", password: "687xvf" }
            ],
        isLoggedIn: false
};
```

IIFE

```
return (
    <div>
        {
            ( () => {

                // Your Code

            } ) ()
        }
    </div>
);
```

In React, we use curly braces to wrap an IIFE, put all the logic you want inside it (if/else, switch, ternary operators, etc), and return whatever you want to render.

Inline if with Logical && Operator

You may embed any expressions in JSX by wrapping them in curly braces. This includes the JavaScript logical && operator.

Operand 1 &&	Operand 2	Result
True	True	True
True	False	False
False	True	False
False	False	False
True	Expression	Expression
False	Expression	False

Ex:- purchase && <Payment />
If *purchase* evaluates to *true*, the <Payment /> component will be return

Ex:- purchase && <Payment />
If *purchase* evaluates to *false*, the <Payment /> component will be ignored

true && expresion1 && expression2 = expression2

Conditional Rendering

Conditional rendering in React works the same way conditions work in JavaScript.

Use JavaScript operators like if or the conditional (ternary) operator to create elements representing the current state, and let React update the UI to match them.

if and if-else statements don't work inside JSX. This is because JSX is just syntactic sugar for function calls and object construction.

```
<div id={if (condition) { 'msg' }}>Hello</div>
```

```
React.createElement("div", {id: if (condition) { 'msg' }}, "Hello");
```

Using Custom Hook

A custom Hook is a JavaScript function whose name starts with ”use” and that may call other Hooks.

Ex:-

```
const data = useSomething( );
```

Creating Custom Hook

A custom Hook is a JavaScript function whose name starts with ”use” and that may call other Hooks.

Ex:-

```
function useSomething( ) {
        return
};
```

Custom Hook

A custom Hook is a JavaScript function, when we want to share logic between two JavaScript functions, we extract it to a third function.

Building your own Hooks lets you extract component logic into reusable functions.

You can write custom Hooks that cover a wide range of use cases like form handling, animation, declarative subscriptions, timers, and many more.

Does useEffect run after every Render

Yes! By default, it runs both after the first render and after every update.

Why is useEffect called inside a Component

Placing useEffect inside the component lets us access the state variable or any props right from the effect.

What does useEffect do ?

By using this Hook, you tell React that your component needs to do something after render. React will remember the function you passed and call it later after performing the DOM updates. In this effect, we set the document title, we could also perform data fetching or call some other imperative API.

useEffect ( )

```
useEffect(() => {
    console.log("Hello useEffect");
});

useEffect(() => {
    console.log("Hello useEffect");
}, [count]);
```

useEffect ( )

useEffect is a hook for encapsulating code that has 'side effects,' if you're familiar with React class lifecycle methods, you can think of useEffect Hook as componentDidMount, componentDidUpdate, and componentWillUnmount combined.

Ex:-

```
import React, { useState, useEffect } from 'react';
useEffect(Function)
useEffect(Function, Array)
```

importing useEffect Hook from React

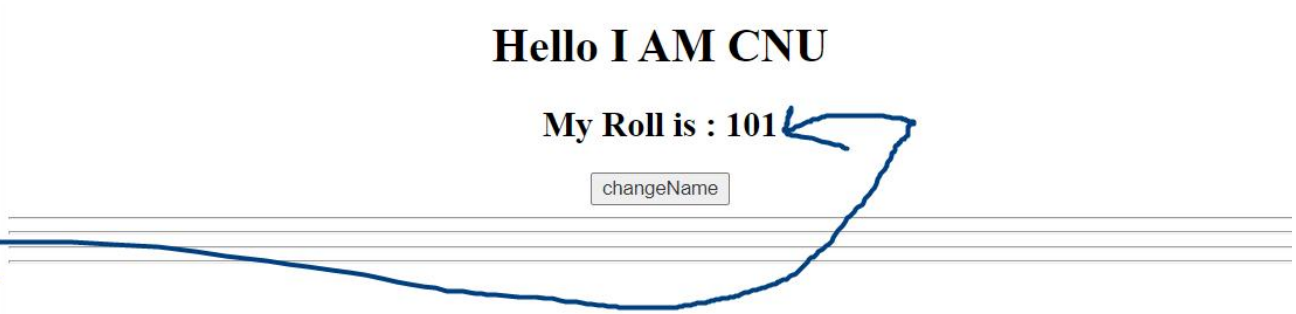
- The function passed to useEffect will run after the render is committed to the screen.
- Second argument to useEffect that is the array of values that the effect depends on.

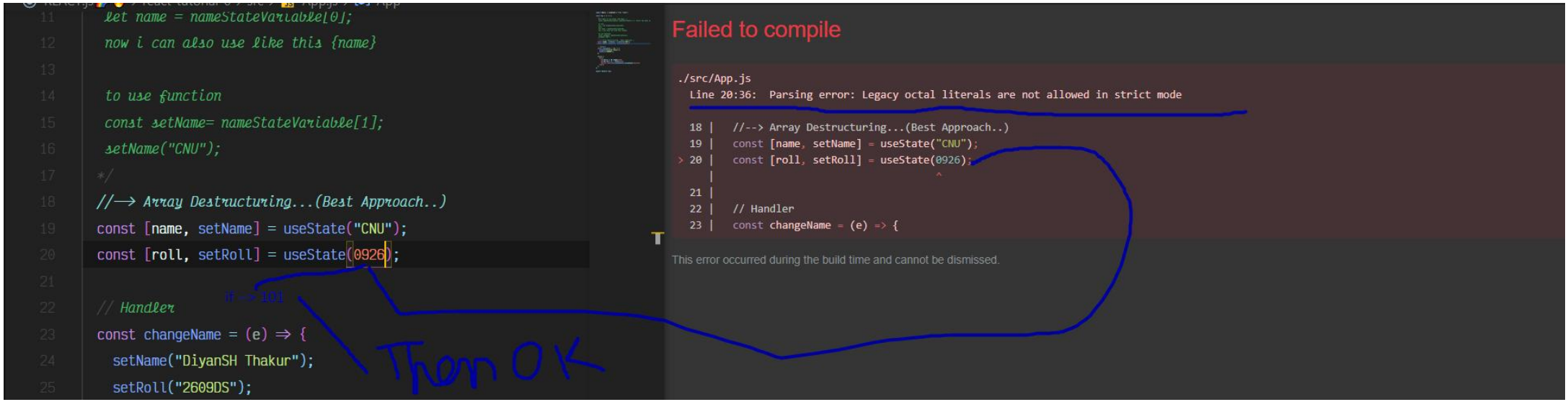
Note - you can call useEffect as many times as you want.

Effect Hooks

The Effect Hook lets you perform side effects in function components. Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects.

```
let name = nameStateVariable[0];
now i can also use like this {name}

to use function
const setName= nameStateVariable[1];
setName("CNU");
*/
//—> Array Destructuring...(Best Approach..)
const [name, setName] = useState("CNU");
const [roll, setRoll] = useState(101);
```




let name = nameStateVariable[0];
now i can also use like this {name}
to use function
const setName= nameStateVariable[1];
setName("CNU");
*/
//--> Array Destructuring...(Best Approach..)
const [name, setName] = useState("CNU");
const [roll, setRoll] = useState(0926);
if --> 101
// Handler
const changeName = (e) => {
setName("DiyanSH Thakur");
setRoll("2609DS");
Then OK
Failed to compile
./src/App.js
Line 20:36: Parsing error: Legacy octal literals are not allowed in strict mode
18 | //--> Array Destructuring...(Best Approach..)
19 | const [name, setName] = useState("CNU");
> 20 | const [roll, setRoll] = useState(0926);
21 |
22 | // Handler
23 | const changeName = (e) => {
This error occurred during the build time and cannot be dismissed.

```
function App () {
const nameStateVariable = useState("Rahul");
const [name, setName] = useState("Rahul");
const [roll, setRoll] = useState(101);
const [subject, setSubject] = useState( [  {sub: "Math"}  ] );
}
```

Declaring State

```
const [name, setName] = useState("Rahul");
```

Declaring State Variable

```
const [roll, setRoll] = useState(101);
```

When we declare a state variable with useState, it returns a pair - an array with two items. So, by writing square bracket we are doing Array Destructuring.

```
const nameStateVariable = useState("Rahul");
```

- The first item is the current value.
- The second is a function that lets us update it.

```
const name = nameStateVariable[0];          // First item of Array
const setName = nameStateVariable[1];       // Second item of Array
```

Note - you can call useState as many times as you want.

Declaring State

```
const [name, setName] = useState("Rahul");
```

When we declare a state variable with useState, it returns a pair - an array with two items. So, by writing square bracket we are doing Array Destructuring.

```
const nameStateVariable = useState("Rahul");
```

- The first item is the current value.
- The second is a function that lets us update it.

```
const name = nameStateVariable[0];          // First item of Array
const setName = nameStateVariable[1];       // Second item of Array
```

Declaring State

useState ( ) - useState is a Hook that allows you add React state to function components. We call it inside a function component to add some local state to it.

useState returns a pair - the current state value and a function that lets you update it.

React will preserve this state between re-renders.

You can call this function from an event handler or somewhere else.

Ex:-

```
import React, { useState } from 'react';
const nameStateVariable = useState("Rahul");
const [name, setName] = useState("Rahul");
```

importing useState Hook from React

Declaring State Variable

Rules of Hooks

- Only call Hooks at the top level – We should not call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function.
- Only call Hooks from React functions – We should not call Hooks from regular JavaScript functions. Instead, call Hooks from React function components or call Hooks from custom Hooks
- React relies on the order in which Hooks are called.
- Hooks don’t work inside classes.

Hooks

Hooks are functions that let you "hook into" React state and lifecycle features from function components.

Hooks allow you to use React without classes. It means you can use state and other React features without writing a class.

React provides a few built-in Hooks like useState, useEffect etc

Hooks are a new addition in React 16.8.

When use Hooks

If you write a function component and realize you need to add some state to it.

```
class Container extends React.Component {
  constructor(props) {
    super(props);
    this.state = {show: true};
  }
  delHeader = () => {
    this.setState({show: false});
  }
  render() {
    let myheader;
    if (this.state.show) {
      myheader = <Child />;
    };
    return (
      <div>
      {myheader}
      <button type="button" onClick={this.delHeader}>Delete Header</button>
      </div>
    );
  }
}

class Child extends React.Component {
  componentWillUnmount() {
    alert("The component named Header is about to be unmounted.");
  }
  render() {
    return (
      <h1>Hello World!</h1>
    );
  }
}

ReactDOM.render(<Container />, document.getElementById('root'));
```

componentWillUnmount

The `componentWillUnmount` method is called when the component is about to be removed from the DOM.

Unmounting

The next phase in the lifecycle is when a component is removed from the DOM, or *unmounting* as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

- `componentWillUnmount()`

The `componentDidUpdate` method is called after the update has been rendered in the DOM:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  componentDidUpdate() {
    document.getElementById("mydiv").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <div id="mydiv"></div>
      </div>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

componentDidUpdate

The `componentDidUpdate` method is called after the component is updated in the DOM.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted,* a timer changes the state, and the color becomes "yellow".

This action triggers the *update* phase, and since this component has a `componentDidUpdate` method, this method is executed and writes a message in the empty DIV element:

componentDidUpdate

The `componentDidUpdate` method is called after the component is updated in the DOM.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted,* a timer changes the state, and the color becomes "yellow".

This action triggers the *update* phase, and since this component has a `componentDidUpdate` method, this method is executed and writes a message in the empty DIV element:

Use the `getSnapshotBeforeUpdate()` method to find out what the `state` object looked like before the update:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    document.getElementById("div1").innerHTML =
    "Before the update, the favorite was " + prevState.favoritecolor;
  }
  componentDidUpdate() {
    document.getElementById("div2").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
        <div id="div1"></div>
        <div id="div2"></div>
      </div>
    );
  }
}
```

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.

This is executed just before the component gets removed from the DOM.

Syntax:-

```
componentWillUnmount(){

}
```

getSnapshotBeforeUpdate

In the `getSnapshotBeforeUpdate()` method you have access to the `props` and `state` *before* the update, meaning that even after the update, you can check what the values were *before* the update.

If the `getSnapshotBeforeUpdate()` method is present, you should also include the `componentDidUpdate()` method, otherwise you will get an error.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted,* a timer changes the state, and after one second, the favorite color becomes "yellow".

This action triggers the *update* phase, and since this component has a `getSnapshotBeforeUpdate()` method, this method is executed, and writes a message to the empty DIV1 element.

Then the `componentDidUpdate()` method is executed and writes a message in the empty DIV2 element:

render

The `render()` method is of course called when a component gets *updated*, it has to re-render the HTML to the DOM, with the new changes.

The example below has a button that changes the favorite color to blue:

Example:

Click the button to make a change in the component's state:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

componentDidUpdate()

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

This method is used to re trigger the third party libraries used to make sure these libraries also update and reload themselves.

componentDidUpdate() will not be invoked if shouldComponentUpdate() returns false.

Syntax:-

```
componentDidUpdate(prevProps, prevState, snapshot) {


}
```

If your component implements the getSnapshotBeforeUpdate() lifecycle (which is rare), the value it returns will be passed as a third "snapshot" parameter to componentDidUpdate(). Otherwise this parameter will be undefined.

getSnapshotBeforeUpdate()

This method is called right before the virtual DOM is about to make change to the DOM (before DOM is updated), which allows our components to capture the current values or some information from the DOM(eg. Scroll Position) before it is potentially changed. Any value returned by this lifecycle will be passed as third parameter to componentDidUpdate() .

Syntax: -

```
getSnapshotBeforeUpdate(prevProps, prevState){

}
```

Stop the component from rendering at any update:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  shouldComponentUpdate() {
    return false;
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

shouldComponentUpdate

In the `shouldComponentUpdate()` method you can return a Boolean value that specifies whether React should continue with the rendering or not.

The default value is `true`.

The example below shows what happens when the `shouldComponentUpdate()` method returns `false`:

shouldComponentUpdate()

Use shouldComponentUpdate() to let React know if a component's output is not affected by the current change in state or props, It means should React re-render or it can skip rendering? shouldComponentUpdate() is invoked before rendering when new props or state are being received. This method return true by default.

render() will not be invoked if shouldComponentUpdate() returns false.

Syntax: -

```
shouldComponentUpdate(nextProps, nextState){

}
```

Updating

The next phase in the lifecycle is when a component is *updated*.

A component is updated whenever there is a change in the component's `state` or `props`.

React has five built-in methods that gets called, in this order, when a component is updated:

1. `getDerivedStateFromProps()`
2. `shouldComponentUpdate()`
3. `render()`
4. `getSnapshotBeforeUpdate()`
5. `componentDidUpdate()`

The `render()` method is required and will always be called, the others are optional and will be called if you define them.

componentDidMount

The `componentDidMount()` method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

Example:

At first my favorite color is red, but give me a second, and it is yellow instead:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

Updating

Updating – Updating is the process of changing state or porps of component and update changes to nodes already in the DOM.

An update can be caused by changes to props or state. These methods are called in the following order when a component is being re-rendered:

- static getDerivedStateFromProps()
- shouldComponentUpdate()
- **render()**
- getSnapshotBeforeUpdate()
- **componentDidUpdate()**

componentDidMount( )

componentDidMount() is invoked immediately after a component is mounted (inserted into the tree), after the render() method has taken place.

This method is executed once in a lifecycle of a component and after the first render.

Initialization that requires DOM nodes should go here.

This is where AJAX requests and DOM or state updates should occur. This method is also used for integration with other JavaScript frameworks and any functions with delayed execution such as setTimeout or setInterval.

The API calls should be made in componentDidMount method always.

Syntax:-

```
componentDidMount() {


}
```

componentDidMount( )

componentDidMount() is invoked immediately after a component is mounted (inserted into the tree), after the render() method has taken place.

This method is executed once in a lifecycle of a component and after the first render.

Initialization that requires DOM nodes should go here.

This is where AJAX requests and DOM or state updates should occur. This method is also used for integration with other JavaScript frameworks and any functions with delayed execution such as setTimeout or setInterval.

The API calls should be made in componentDidMount method always.

componentDidMount

The `componentDidMount()` method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

Example:

At first my favorite color is red, but give me a second, and it is yellow instead:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

render( )

The render() method is the only required method in a class component. It examines this.props and this.state .

It returns one of the following types:

React elements – These are created via JSX(Not required).

For example, and <App /> are React elements that instruct React to render a DOM node, or another user-defined component, respectively.

Arrays and fragments - It is used to return multiple elements from render.

Portals – It is used to render children into a different DOM subtree.

String and numbers - These are rendered as text nodes in the DOM.

Booleans or null - It renders nothing. (Mostly exists to support return test && <Child /> pattern, where test is boolean.)

Note - The render() function should be pure, meaning that it does not modify component state, it returns the same result each time it's invoked, and it does not directly interact with the browser.

render

The `render()` method is required, and is the method that actually outputs the HTML to the DOM.

Example:

A simple component with a simple `render()` method:

```
class Header extends React.Component {
  render() {
    return (
      <h1>This is the content of the Header component</h1>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

The `getDerivedStateFromProps` method is called right before the render method:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));
```

getDerivedStateFromProps

The `getDerivedStateFromProps()` method is called right before rendering the element(s) in the DOM.

This is the natural place to set the `state` object based on the initial `props`.

It takes `state` as an argument, and returns an object with changes to the `state`.

The example below starts with the favorite color being "red", but the `getDerivedStateFromProps()` method updates the favorite color based on the `favcol` attribute:

static getDerivedStateFromProps( )

getDerivedStateFromProps is invoked right before calling the render method, both on the initial mount and on subsequent updates. It should return an object to update the state, or null to update nothing. This method exists for rare use cases where the state depends on changes in props over time. This method doesn't have access to the component instance.

Syntax:-

```
static getDerivedStateFromProps(props, state) {

}
```

Rarely Used

```
constructor(props) {
    super(props);
    this.state = {
      name: "Rahul",
      roll: this.props.roll
    };
    this.handleClick = this.handleClick.bind(this);
  }
```

If you don't initialize state and you don't bind methods, you don't need to implement a constructor for your React component.

React constructors are only used for two purposes:

- Initializing local state by assigning an object to this.state.

  Ex:- this. State = {name: “Rahul”}

- Binding event handler methods to an instance.

  Ex:- this.handleClick = this.handleClick.bind(this);

constructor

The `constructor()` method is called before anything else, when the component is initiated, and it is the natural place to set up the initial `state` and other initial values.

The `constructor()` method is called with the `props`, as arguments, and you should always start by calling the `super(props)` before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (`React.Component`).

Example:

The `constructor` method is called, by React, every time you make a component:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

Mounting

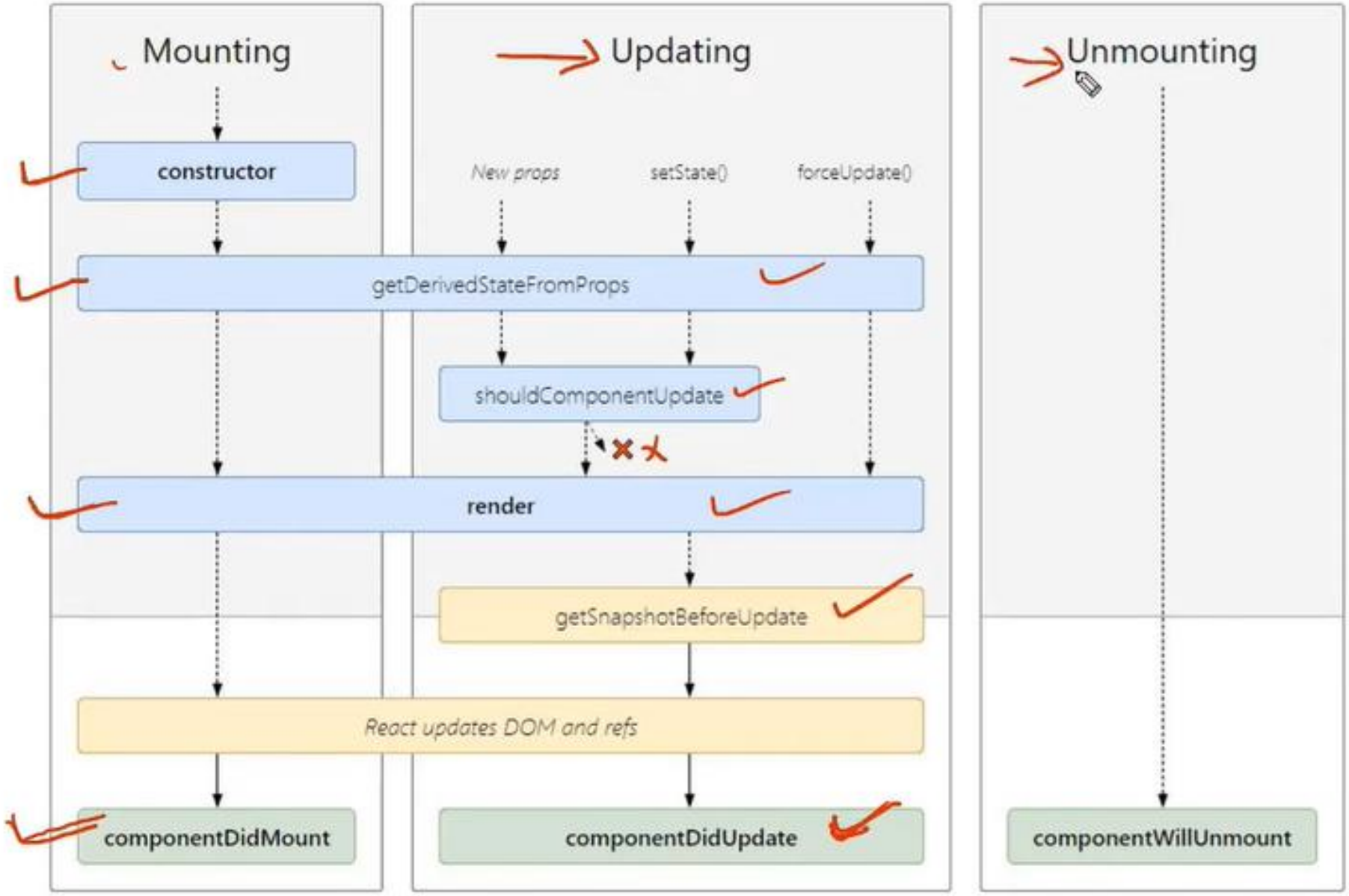
Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

1. `constructor()`
2. `getDerivedStateFromProps()`
3. `render()`
4. `componentDidMount()`

The `render()` method is required and will always be called, the others are optional and will be called if you define them.

Lifecycle Methods



Lifecycle Methods

Each component has several "lifecycle methods" that you can override to run code at particular times in the process.

- **Mounting**
- **Updating**
- Error Handling
- **Unmounting**

Methods which each component runs during each of its phases.

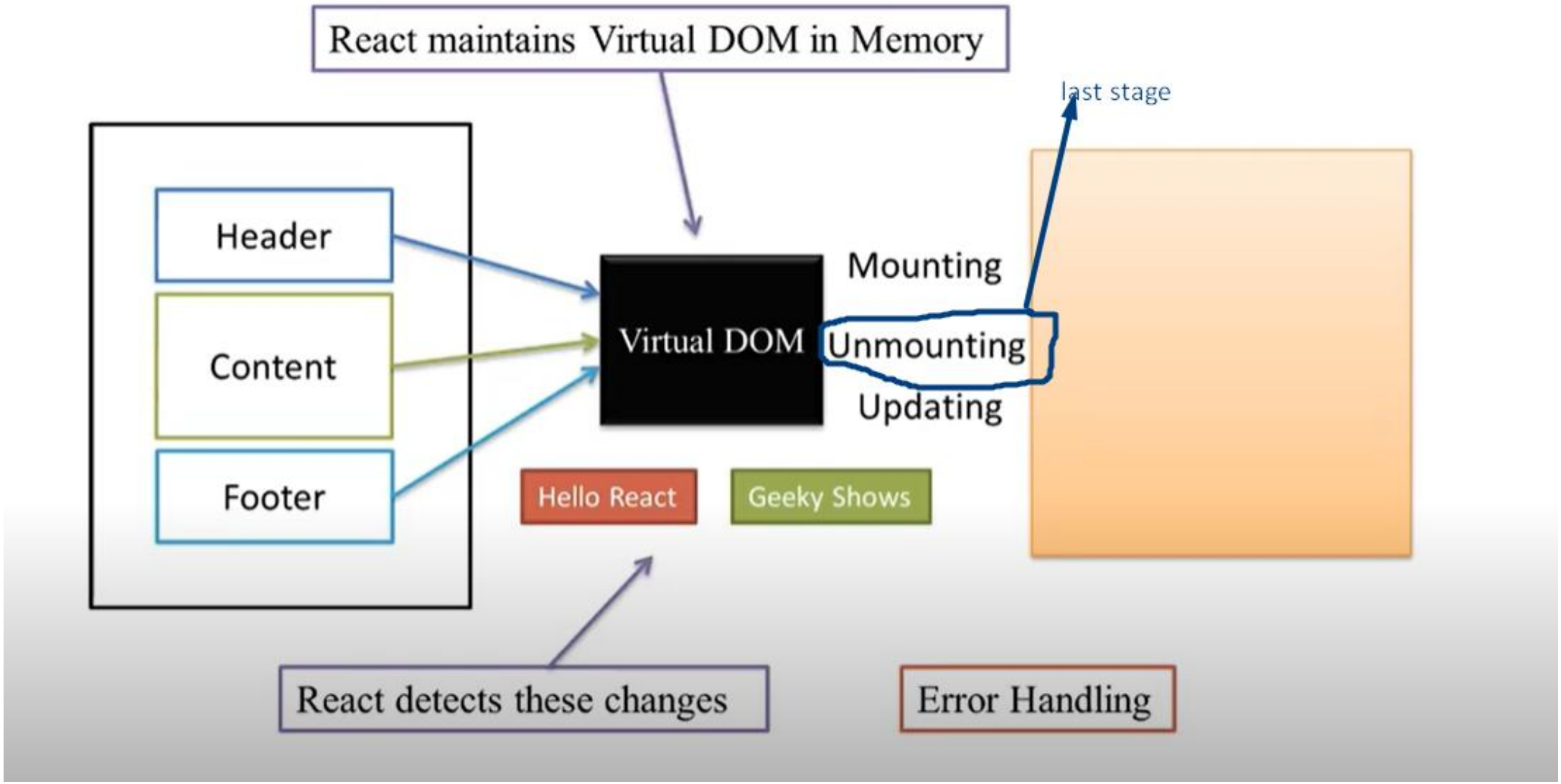
Lifecycle of Components

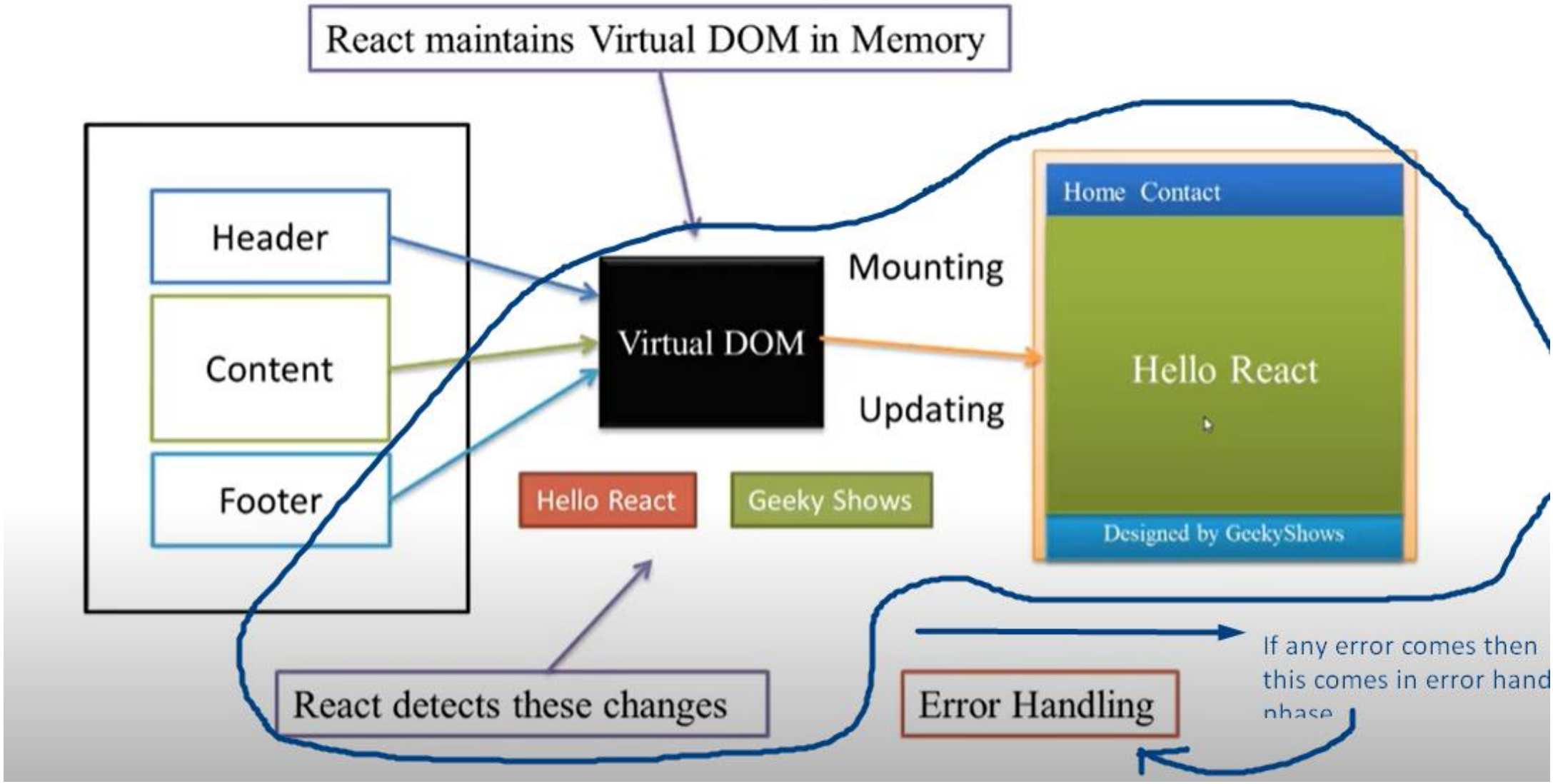
Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

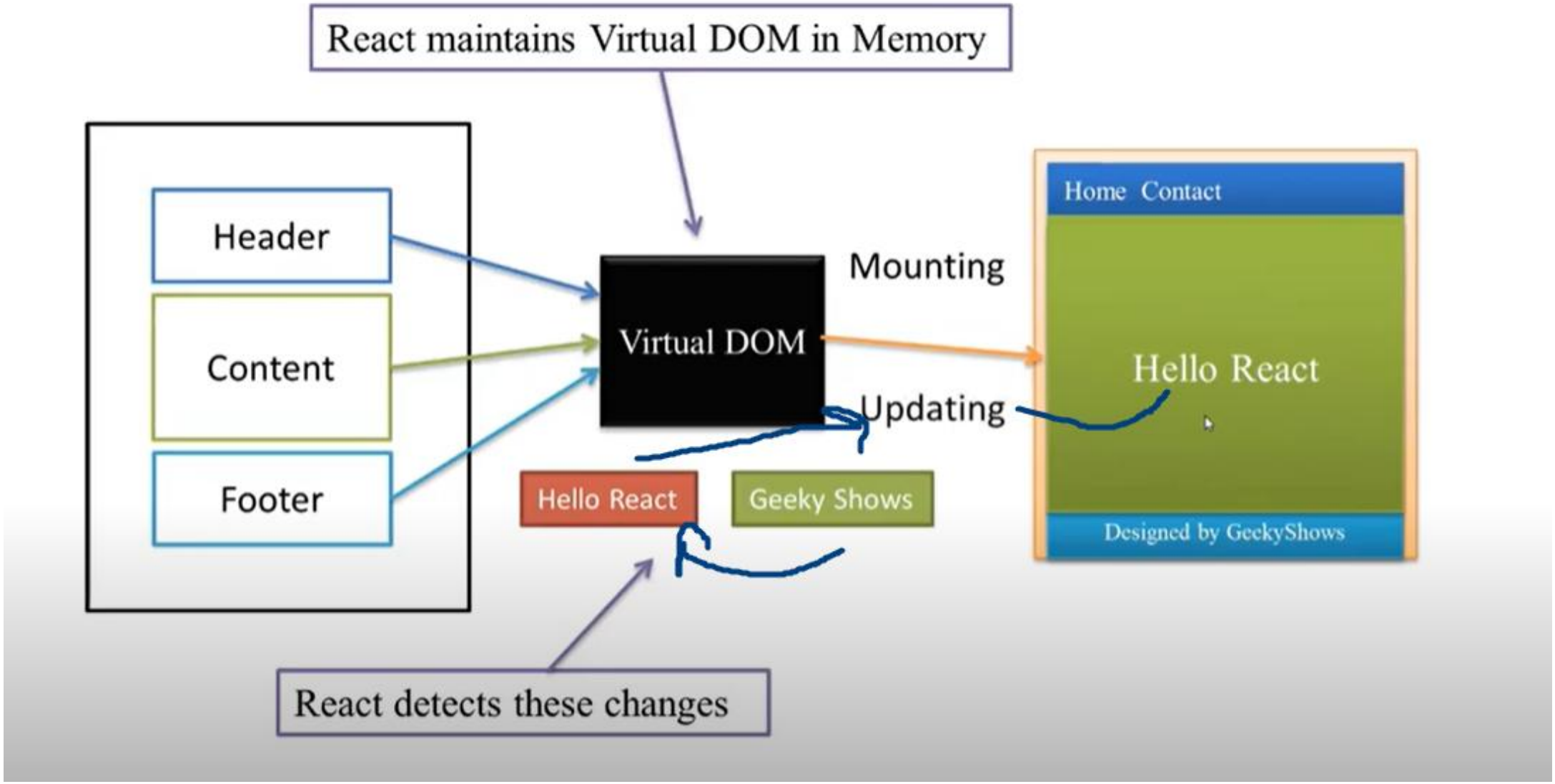
The three phases are: **Mounting**, **Updating**, and **Unmounting**.

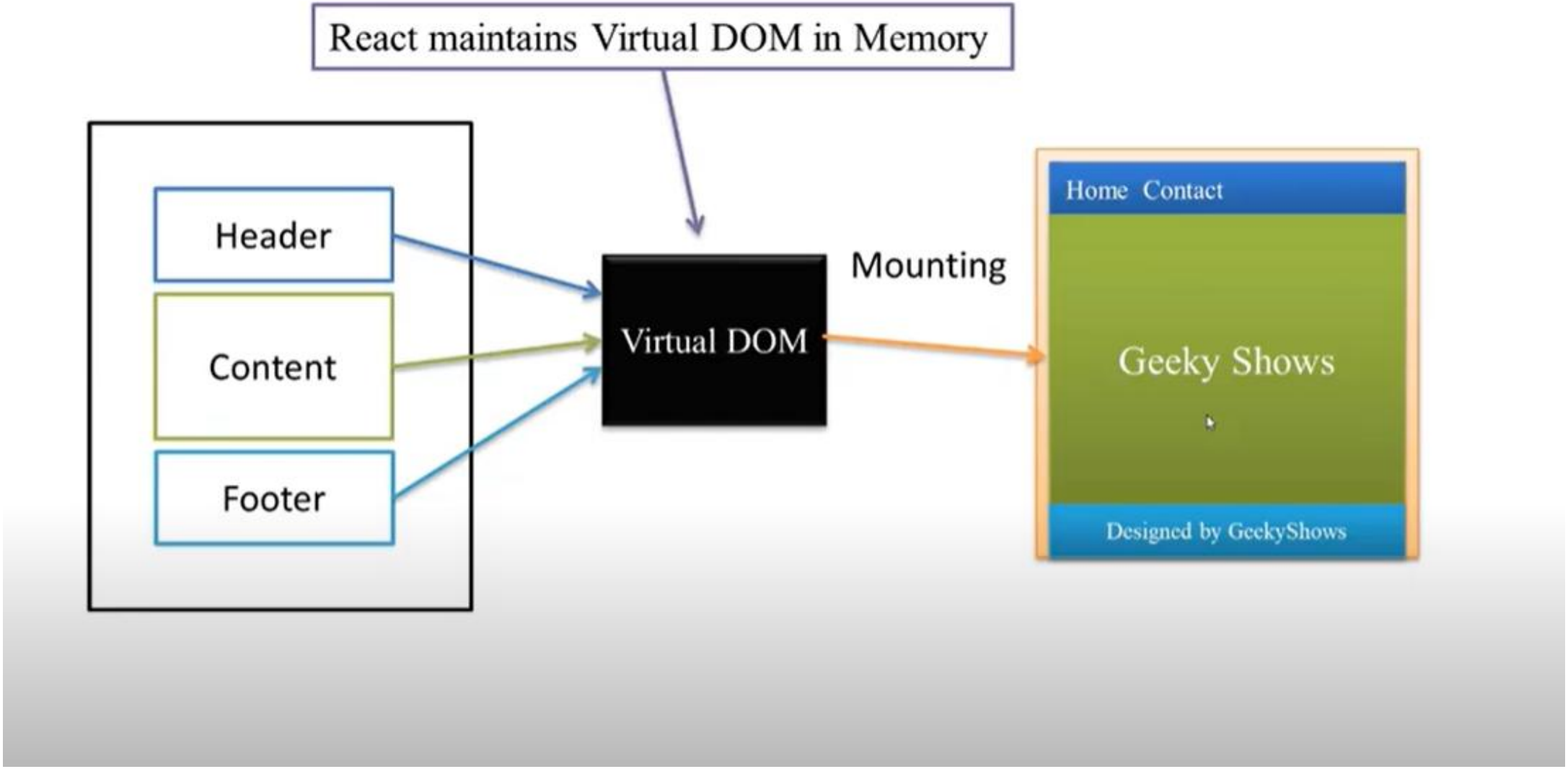
The primary difference between the BA and the BM is that the BM is a pre-professional degree, preparing students for careers in performance, education, or pedagogy, while the BA is a liberal arts degree, designed to develop broadly-educated musicians and citizens. In terms of course requirements, the most important differences are as follows:

- BA students must demonstrate competency in foreign language, typically by taking four semesters of a single foreign language (see other options).
- BA students have less requirements in private lessons, ensemble, music theory, aural skills, pedagogy, and literature, and more music electives and general electives.
- BA students must take at least one class in the department of Art and Design or the department of Theatre.
- BM students must do a senior recital; BA students may design their own capstone project.


React maintains Virtual DOM in Memory
last stage
Header
Content
Footer
Virtual DOM
Mounting
Unmounting
Updating
Hello React
Geeky Shows
React detects these changes
Error Handling


React maintains Virtual DOM in Memory
Header
Content
Footer
Virtual DOM
Mounting
Updating
Home Contact
Hello React
Designed by GeekyShows
Hello React
Geeky Shows
React detects these changes
Error Handling
If any error comes then
this comes in error hand
nhase


React maintains Virtual DOM in Memory
Header
Content
Footer
Virtual DOM
Mounting
Updating
Hello React
Geeky Shows
Home Contact
Hello React
Designed by GeekyShows
React detects these changes


React maintains Virtual DOM in Memory
Header
Content
Footer
Virtual DOM
Mounting
Home Contact
Geeky Shows
Designed by GeekyShows

Phase of Component

- Mounting - Mounting is the process of creating an element and inserting it in a DOM tree.
- Updating – Updating is the process of changing state or porps of component and update changes to nodes already in the DOM.
- Error Handling – These are used when there is error during rendering, in lifecycle method or in the constructor of any child component.
- Unmounting – Unmounting is the process of removing components from the DOM.

Passing Arguments to Event Handlers

- **Arrow Function**

```
<button onClick={(e) => this.handleClick(id, e)}>Delete</button>
```

- **Bind Method**

```
<button onClick={this.handleClick.bind(this, id)}>Delete</button>
```

Note:-

- In both cases, the ***e*** argument representing the React event will be passed as a second argument after the ID.
- With an arrow function, we have to pass it explicitly, but with bind any further arguments are automatically forwarded.

Update State

```
this.setState(function(state, props) {
  return {

  };
});
```

- It accepts a function rather than an object.
- It receives the previous state as the first argument,
- The props at the time the update is applied as the second argument.

2. Bind the event handler to `this`.

Note that the first argument has to be `this`.

Example:

Send "Goal" as a parameter to the `shoot` function:

```
class Football extends React.Component {
  shoot(a) {
    alert(a);
  }
  render() {
    return (
      <button onClick={this.shoot.bind(this, "Goal")}>Take the shot!</button>
    );
  }
}

ReactDOM.render(<Football />, document.getElementById('root'));
```

Passing Arguments

If you want to send parameters into an event handler, you have two options:

1. Make an anonymous arrow function:

Example:

Send "Goal" as a parameter to the `shoot` function, using arrow function:

```
class Football extends React.Component {
  shoot = (a) => {
    alert(a);
  }
  render() {
    return (
      <button onClick={() => this.shoot("Goal")}>Take the shot!</button>
    );
  }
}

ReactDOM.render(<Football />, document.getElementById('root'));
```

Event Handling

You cannot return *false* to prevent default behavior in React. You must call *preventDefault* explicitly.

In HTML

```
<a href="#" onclick="console.log('Clicked.'); return false"> Click me </a>
```

In React

```
function handleClick(e) {
   e.preventDefault();
   console.log(Clicked.');
 }

<a href="#" onClick={handleClick}> Click me </a>
```

Add a button with an `onClick` event that will change the color property:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  changeColor = () => {
    this.setState({color: "blue"});
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
        <button
          type="button"
          onClick={this.changeColor}
        >Change color</button>
      </div>
    );
  }
}
```

Changing the `state` Object

To change a value in the state object, use the `this.setState()` method.

When a value in the `state` object changes, the component will re-render, meaning that the output will change according to the new value(s).

Using the `state` Object

Refer to the `state` object anywhere in the component by using the `this.state.propertyname` syntax:

Example:

Refer to the `state` object in the `render()` method:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
      </div>
    );
  }
}
```

Creating the `state` Object

The `state` object is initialized in the constructor:

Example:

Specify the `state` object in the constructor method:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {brand: "Ford"};
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
    );
  }
}
```

React components has a built-in `state` object.

The `state` object is where you store property values that belongs to the component.

When the `state` object changes, the component re-renders.

Props in the Constructor

If your component has a constructor function, the props should always be passed to the constructor and also to the React.Component via the `super()` method.

Example

```
class Car extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return <h2>I am a {this.props.model}!</h2>;
  }
}

ReactDOM.render(<Car model="Mustang"/>, document.getElementById('root'));
```

Note: React Props are read-only! You will get an error if you try to change their value.

If you have a variable to send, and not a string as in the example above, you just put the variable name inside curly brackets:

Example

Create a variable named "carname" and send it to the Car component:

```
class Car extends React.Component {
  render() {
    return <h2>I am a {this.props.brand}!</h2>;
  }
}

class Garage extends React.Component {
  render() {
    const carname = "Ford";
    return (
      <div>
      <h1>Who lives in my garage?</h1>
      <Car brand={carname} />
      </div>
    );
  }
}

ReactDOM.render(<Garage />, document.getElementById('root'));
```

Pass Data

Props are also how you pass data from one component to another, as parameters.

Example

Send the "brand" property from the Garage component to the Car component:

```
class Car extends React.Component {
  render() {
    return <h2>I am a {this.props.brand}!</h2>;
  }
}

class Garage extends React.Component {
  render() {
    return (
      <div>
      <h1>Who lives in my garage?</h1>
      <Car brand="Ford" />
      </div>
    );
  }
}

ReactDOM.render(<Garage />, document.getElementById('root'));
```

Add a "brand" attribute to the Car element:

```
const myelement = <Car brand="Ford" />;
```

The component receives the argument as a `props` object:

Example

Use the brand attribute in the component:

```
class Car extends React.Component {
  render() {
    return <h2>I am a {this.props.brand}!</h2>;
  }
}
```

Props are arguments passed into React components.

Props are passed to components via HTML attributes.

React Props

React Props are like function arguments in JavaScript *and* attributes in HTML.

To send props into a component, use the same syntax as HTML attributes:

Components in Components

We can refer to components inside other components:

Example

Use the Car component inside the Garage component:

```
class Car extends React.Component {
  render() {
    return <h2>I am a Car!</h2>;
  }
}

class Garage extends React.Component {
  render() {
    return (
      <div>
      <h1>Who lives in my Garage?</h1>
      <Car />
      </div>
    );
  }
}

ReactDOM.render(<Garage />, document.getElementById('root'));
```

```
class Car extends React.Component {
  constructor() {
    super();
    this.state = {color: "red"};
  }
  render() {
    return <h2>I am a {this.state.color} Car!</h2>;
  }
}
```

Create a constructor function in the Car component, and add a color property:

```
class Car extends React.Component {
  constructor() {
    super();
    this.state = {color: "red"};
  }
  render() {
    return <h2>I am a Car!</h2>;
  }
}
```

Component Constructor

If there is a `constructor()` function in your component, this function will be called when the component gets initiated.

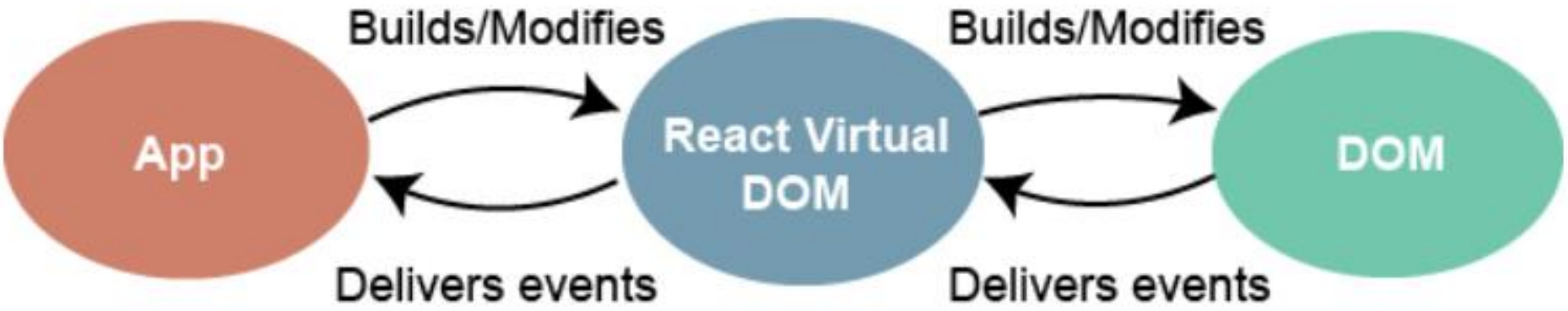
The constructor function is where you initiate the component's properties.

In React, component properties should be kept in an object called `state`.

You will learn more about `state` later in this tutorial.

The constructor function is also where you honor the inheritance of the parent component by including the `super()` statement, which executes the parent component's constructor function, and your component has access to all the functions of the parent component (`React.Component`).

Events Handler



Another difference is that you cannot return `false` to prevent default behavior in React. You must call `preventDefault` explicitly. For example, with plain HTML, to prevent the default link behavior of opening a new page, you can write:

```
<a href="#" onclick="console.log('The link was clicked.'); return false">
  Click me
</a>
```

In React, this could instead be:

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

For example, the HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

is slightly different in React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Why Arrow Functions?

In class components, the `this` keyword is not defined by default, so with regular functions the `this` keyword represents the object that called the method, which can be the global window object, a HTML button, or whatever.

Read more about binding `this` in our React ES6 'What About this?' chapter.

If you *must* use regular functions instead of arrow functions you have to bind `this` to the component instance using the `bind()` method:

The Root Node

The root node is the HTML element where you want to display the result.

It is like a *container* for content managed by React.

It does NOT have to be a `<div>` element and it does NOT have to have the `id='root'` :

Example

The root node can be called whatever you like:

```
<body>

  <header id="sandy"></header>

</body>
```

Display the result in the `<header id="sandy">` element:

```
ReactDOM.render(<p>Hallo</p>, document.getElementById('sandy'));
```

What About `this`?

The handling of `this` is also different in arrow functions compared to regular functions.

In short, with arrow functions there are no binding of `this`.

In regular functions the `this` keyword represented the object that called the function, which could be the window, the document, a button or whatever.

With arrow functions, the `this` keyword *always* represents the object that defined the arrow function.

Let us take a look at two examples to understand the difference.

Both examples call a method twice, first when the page loads, and once again when the user clicks a button.

The first example uses a regular function, and the second example uses an arrow function.

The result shows that the first example returns two different objects (window and button), and the second example returns the Header object twice.

Event Handling

Handling events with React elements is very similar to handling events on DOM elements. There are some syntactic differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

In HTML

```
<button onclick="handleClick()">Click Me</button>
```

In React

```
<button onClick={handleClick}>Click Me</button>          // Function Component
<button onClick={this.handleClick}>Click Me</button>     // Class Component
```

What is Event

The actions to which JavaScript can respond are called Events.

- Clicking an element
- Submitting a form
- Scrolling page
- Hovering an element

Inside the Constructor

```
class Student extends Component {
 constructor(props) {
  // It is required to call the parent class constructor
  super(props);

  // States
  this.state = {
   name: "Rahul",
   prop1: this.props.prop1
  }
 }
 render() {
 }
}
```

- When the component class is created, the constructor is the first method called, so it's the right place to add state.
- The class instance has already been created in memory, so you can use *this* to set properties on it.
- When we write a constructor, make sure to call the parent class' constructor by super(props)
- When you call super with props. React will make props available across the component through this.props

Directly inside class

```
class Student extends Component {
 // States  - Here it is a class property
state = {
   name: "Rahul",
   prop1: this.props.prop1
 }
 render() {

 }
}
```

Note -

The state property is referred as state.

This is a class instance property.

State

State is similar to props, but it is private and fully controlled by the component. We can create state only in class component. It is possible to update the state/Modify the state.

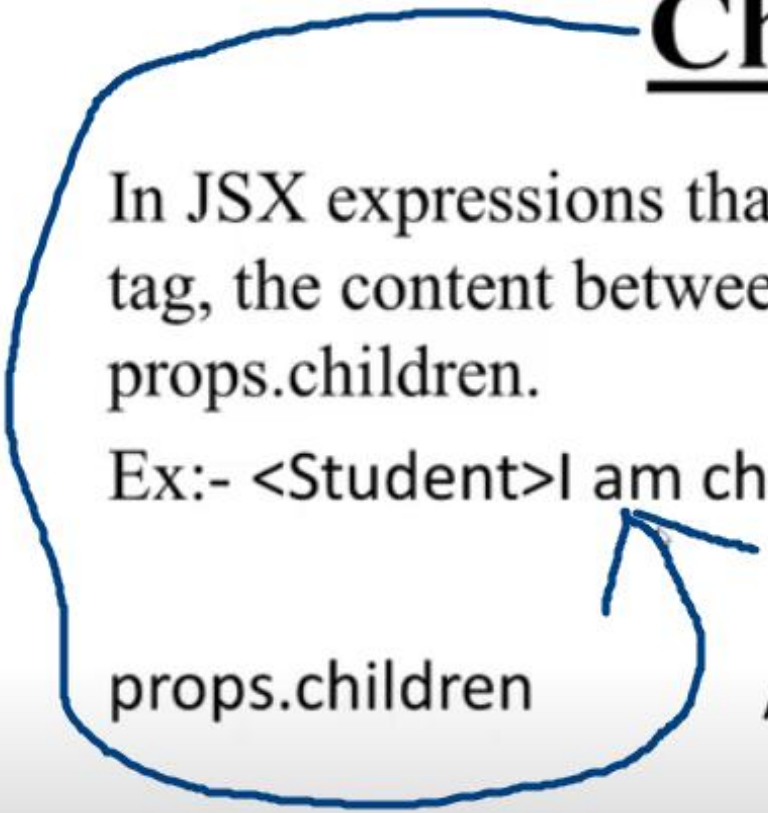
There are two way to initialize state in React Component :-

- Directly inside class
- Inside the constructor

Children in JSX

In JSX expressions that contain both an opening tag and a closing tag, the content between those tags is passed as a special prop: props.children.

Ex:- <Student>I am child</Student>



```
props.children          // I am child
```

Required

```
import PropTypes from 'prop-types';
Student.propTypes = {
  name: PropTypes.string.isRequired
```

Typechecking With PropTypes

PropTypes exports a range of validators that can be used to make sure the data you receive is valid.

```
optionalArray: PropTypes.array,
optionalBool: PropTypes.bool,
optionalFunc: PropTypes.func,
optionalNumber: PropTypes.number,
optionalObject: PropTypes.object,
optionalString: PropTypes.string,
optionalSymbol: PropTypes.symbol,
```

Typechecking With PropTypes

npm install prop-types

To run typechecking on the props for a component, you can assign the special propTypes property.

Ex:-

```
import PropTypes from 'prop-types';
Student.propTypes = {
  name: PropTypes.string
};
```

Here we are assigning default datatype for the incoming data in the attributes of jsx

Note –

- When an invalid value is provided for a prop, a warning will be shown in the JavaScript console.
- For performance reasons, propTypes is only checked in development mode.

Props

Whether you declare a component as a function or a class, it must never modify its own props.
All React components must act like pure functions with respect to their props.

Pure Function

```
function sum(a, b) {
  return a + b;
}
```

Props are Read-Only

Impure Function

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

Don't modify props like this for ex: if u pass rahul in name attribute under JSX then don't change that value inside your comp

Props

```
class Student extends React.Component {
 render() {
   return ( <div>
             <h1>Hello, {this.props.name}</h1>;
             <h2>Your Roll: {this.props.roll}</h2>;
            </div> );
    }
}
ReactDOM.render(  <Student name="Rahul" roll="101" />,  document.getElementById('root') );
```

Book Cylinder

Your order has been successfully placed.

The Order Reference No is: **1200057800094443**

As per current delivery rate expected date of delivery of your refill booking will be within next 3 working days.

Props

When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object "props".

```
function Student(props) {
  return ( <div>
             <h1>Hello, {props.name}</h1>;
             <h2>Your Roll: {props.roll}</h2>;
           </div> );
}
ReactDOM.render(  <Student name="Rahul" roll="101" />,  document.getElementById('root') );
ReactDOM.render(  <Student name={"Rahul"} roll="101" />,  document.getElementById('root') );
ReactDOM.render(  <Student name="Rahul" roll={100+1} />,  document.getElementById('root') );
```

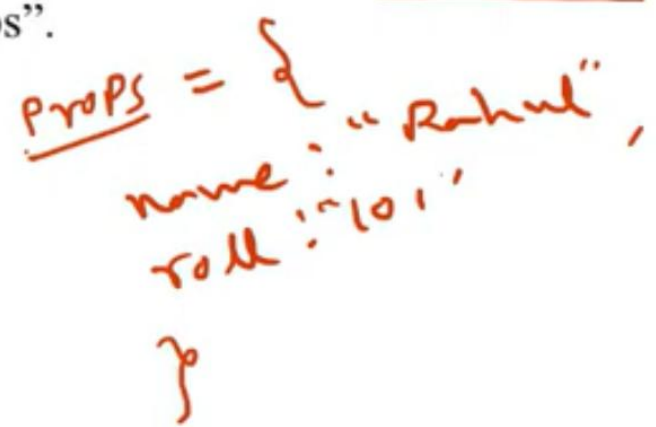
JavaScript Expression as Props

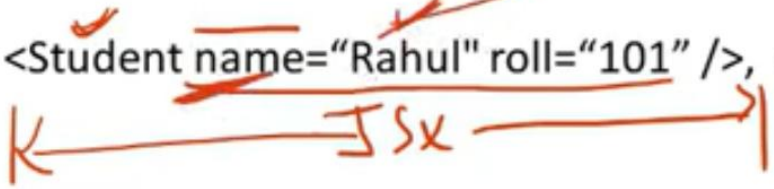
If you pass no value for a prop, it defaults to true

Props

When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object "props".

```
function Student(props) {
  return ( <div>
             <h1>Hello, {props.name}</h1>;
             <h2>Your Roll: {props.roll}</h2>;
           </div> );
}
ReactDOM.render(  <Student name="Rahul" roll="101" />,  document.getElementById('root') );
```

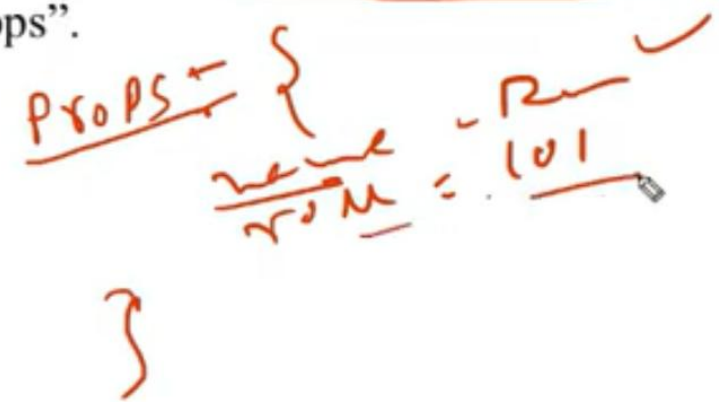




Props

When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object "props".

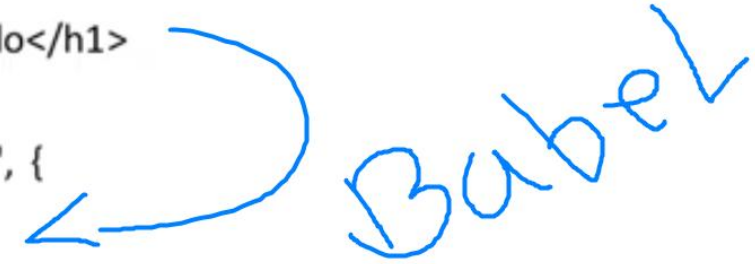
```
function Student(props) {
  return ( <div>
             <h1>Hello, {props.name}</h1>;
             <h2>Your Roll: {props.roll}</h2>;
           </div> );
}
ReactDOM.render( <Student name="Rahul" roll="101" />, document.getElementById('root') );
```



JSX Represents Objects

Babel compiles JSX down to React.createElement() calls.

```
const el = <h1 className="bg">Hello</h1>
```




```
const el = React.createElement("h1", {
  className: "bg"
}, "Hello");

const el = {
  type: 'h1',
  props: {
    className: 'bg',
    children: 'Hello'
  }
};
```

OBJECT REPRESENTAION OF JSX

Specifying Attributes with JSX

You may use quotes to specify string literals as attributes.

Syntax:-

```
const el = <h1 attribute="value"></h1>
```

Ex:-

```
const el = <h1 className="bg">Hello</h1>
const el = <label htmlFor="name">Name</label>
```

You may also use curly braces to embed a JavaScript expression in an attribute.

```
const el = <h1 className={ac.tab}>Hello</h1>
```

```
ReactDOM.render(<App name="Rahul" />, document.getElementById("root"));
```

Note –

- Since JSX is closer to JavaScript than to HTML, React DOM uses camelCase property naming convention instead of HTML attribute names.
- Don't put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

Specifying Attributes with JSX

You may use quotes to specify string literals as attributes.

Syntax:-

```
const el = <h1 attribute="value"></h1>
```

Ex:-

```
const el = <h1 className="bg">Hello</h1>
const el = <label htmlFor="name">Name</label>
```

You may also use curly braces to embed a JavaScript expression in an attribute.

```
const el = <h1 className={ac.tab}>Hello</h1>
```

```
ReactDOM.render(<App name="Rahul" />, document.getElementById("root"));
```

Note –

- Since JSX is closer to JavaScript than to HTML, React DOM uses camelCase property naming convention instead of HTML attribute names.
- Don't put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

JavaScript Expressions in JSX

We can put any valid JavaScript expression inside the curly braces in JSX. You can pass any JavaScript expression as children, by enclosing it within {}.

Syntax:- {expression}

Ex:-

```
const el = <h1>{10+20}</h1>
const el = <h1> Value: {10+20}</h1>

const name = “Rahul”;
const el = <h1>Hello {name}</h1>

const el = <h1>Hello {show()}</h1>
const el = <h1>Hello {user.firstname}</h1>
```

Examples

Ex: -

const el = Hello Rahul	React.createElement("h1", null, "Hello Rahul");
const el = <h1 className="bg">Hello Rahul</h1>	React.createElement("h1", {className: "bg"}, "Hello Rahul");
const el = Hello {name} ;	React.createElement("h1", null, "Hello ", name);
const el = <Student />	React.createElement(Student, null);
const el = <Student name="Rahul" />	React.createElement(Student, {name: "Rahul"});

JavaScript XML (JSX)

JSX stands for JavaScript XML. It is a syntax extension to JavaScript.

JSX is a preprocessor step that adds XML syntax to JavaScript.

JSX produces React "elements". It is possible to create element without JSX but JSX makes React a lot more elegant.

It is recommended to use JSX with React to describe what the UI should look like.

JSX is easier to read and write. Babel transform these expressions into a actual JavaScript Code.

It also allows React to show more useful error and warning messages.

Functional vs Class Component

Use functional components if you are writing a presentational component which doesn't have its own state or needs to access a lifecycle hook. You cannot use setState() in your component because Functional Components are plain JavaScript functions,

Use class Components if you need state or need to access lifecycle hook because all lifecycle hooks are coming from the React.Component which you extend from in class components.

Composing Components

Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail.

Ex:-

```
function Student(){
  return <h1>Hello Rahul</h1>
}
function App( ){
  return (
   <div>
         <Student / >
         <Student / >
        <Student / >
   </div>
);   }
ReactDOM.render(<App />, document.getElementById("root"));
```

Rendering a Component

```
ReactDOM.render(<Student />, document.getElementById("root"));
ReactDOM.render(<Student  name="Rahul"/>, document.getElementById("root"));
```

Ex:-

```
function Student(props){
  return <h1>Hello {props.name}</h1>
}
ReactDOM.render(<Student name="Rahul"/>, document.getElementById("root"));
```

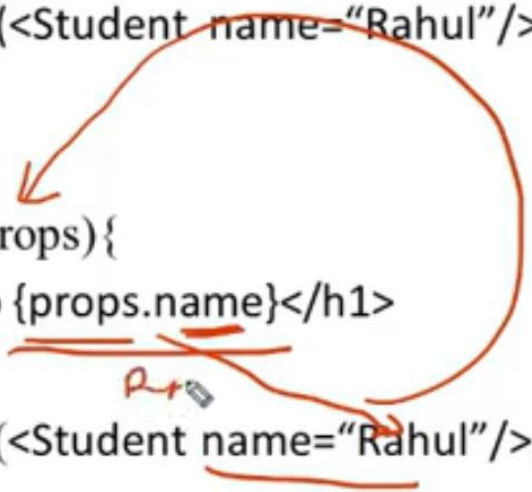
When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object "props".

Rendering a Component

```
ReactDOM.render(<Student />, document.getElementById("root"));
ReactDOM.render(<Student name="Rahul"/>, document.getElementById("root"));
```

Ex:-

```
function Student(props){
  return <h1>Hello {props.name}</h1>
}
ReactDOM.render(<Student name="Rahul"/>, document.getElementById("root"));
```



Class Component

A class component requires you to extend from React.Component. The class must implement a render() member function which returns a React component to be rendered, similar to a return value of a functional component. In a class-based component, props are accessible via this.props.

Syntax:-

```
class class_name extends Component {
  render( ){
      return React Element
   }
}
```

Ex:-

```
class Student extends Component {
  render( ){
      return <h1>Hello Rahul</h1>
   }
}
```

```
class Student extends Component {
  render( ){
      return <h1>Hello {this.props.name}</h1>
   }
}
```

Class Component

A class component requires you to extend from React.Component. The class must implement a render() member function which returns a React component to be rendered, similar to a return value of a functional component. In a class-based component, props are accessible via this.props.

Syntax:-

```
class class_name extends Component {
  render( ){
      return React Element
   }
}
```

Ex:-

```
class Student extends Component {
  render( ){
      return <h1>Hello Rahul</h1>
   }
}
```

Function Components

Syntax:-

```
function func_name (props) {
  return React Element;
}
```

Ex:-

```
function Student(props){
  return <h1>Hello Rahul</h1>
}
function Student(props){
  return <h1>Hello {props.name}</h1>
}
```

```
const Student = (props) =>{
  return <h1>Hello {props.name}</h1>
}
```

Function Components

It is a JavaScript function which accepts a single "props" object argument with data and returns a React Element.

Syntax:-

```
function func_name ( ) {
  return React Element;
}
```

Ex:-

```
function Student( ){
  return <h1>Hello Rahul</h1>
}
```

```
const Student = ( ) =>{
  return <h1>Hello Rahul</h1>
}
```

Components

- Components are the building blocks of any React app.
- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
- Components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.
- Always start component names with a capital letter.
- React treats components starting with lowercase letters as DOM tags. For example, represents an HTML div tag, but <App /> represents a component requires App to be in scope.

React Fragment

Fragment is used to group a list of children without adding extra nodes to the DOM.

Syntax:-

```
<React.Fragment>
</React.Fragment>
```

Ex:-

```
<React.Fragment>
    <h1>Hello</h1>
    <h2>GeekyShows</h2>
</React.Fragment>
```

Syntax:-

```
<>
</>
```

Ex:-

```
<>
    <h1>Hello</h1>
    <h2>GeekyShows</h2>
</>
```

Syntax:-

```
<React.Fragment key={id}>
</React.Fragment>
```

Ex:-

```
<React.Fragment key={item.id}>
   <h1>{item.title}</h1>
    <p>{item.description}</p>
</React.Fragment>
```

ReactDOM.render(element, DOMnode)

ReactDOM.render(element, DOMnode) - It takes a React Element and render it to a DOM node.

Syntax:- ReactDOM.render(element, DOMnode)

- The first argument is which component or element needs to render in the dom.
- The second argument is where to render in the dom.

Ex:-

ReactDOM.render(< App />, document.getElementById("root"));

React.createElement(type, props, children)

React.createElement(type, props, children) - It creates a React Element with the given arguments.

Syntax:- React.createElement(type, props, children)

- type: Type of the html element or component. (example : h1,h2,p,button..etc).
- props: The properties object.

  Example: {style :{ color:“blue”}} or className or event handlers etc.
- children: anything you need to pass between the dom elements.

Ex:-

React.createElement(‘h1’, null, ‘Hello GeekyShows’);

for nested elements we need to write one more react.createElement() method inside this met
like this:
React.createElement('div',null,Reacr.createElement('h1',null,'Hello world'));

React Element

You can create a react element using React.createElement() method but there is a easy way to create element via JSX.

Using createElement() Method

```
React.createElement("h1", null, "Hello GeekyShows");
```

Using JSX

```
<h1>Hello GeekyShows</h1>
```

ultimately this code converts into the above code because browser didn't understand the modern js code hence babel wh
a compiler of jsx, converts this modern is code into browser readable code.

render() Method

The render() method is the only required method in a class component. It examines this.props and this.state .

It returns one of the following types:

React elements – These are created via JSX(Not required).

For example, and <App /> are React elements that instruct React to render a DOM node, or another user-defined component, respectively.

Arrays and fragments - It is used to return multiple elements from render.

Portals – It is used to render children into a different DOM subtree.

String and numbers - These are rendered as text nodes in the DOM.

Booleans or null - It renders nothing. (Mostly exists to support return test && <Child /> pattern, where test is boolean.)

Note - The render() function should be pure, meaning that it does not modify component state, it returns the same result each time it's invoked, and it does not directly interact with the browser.