Chapters

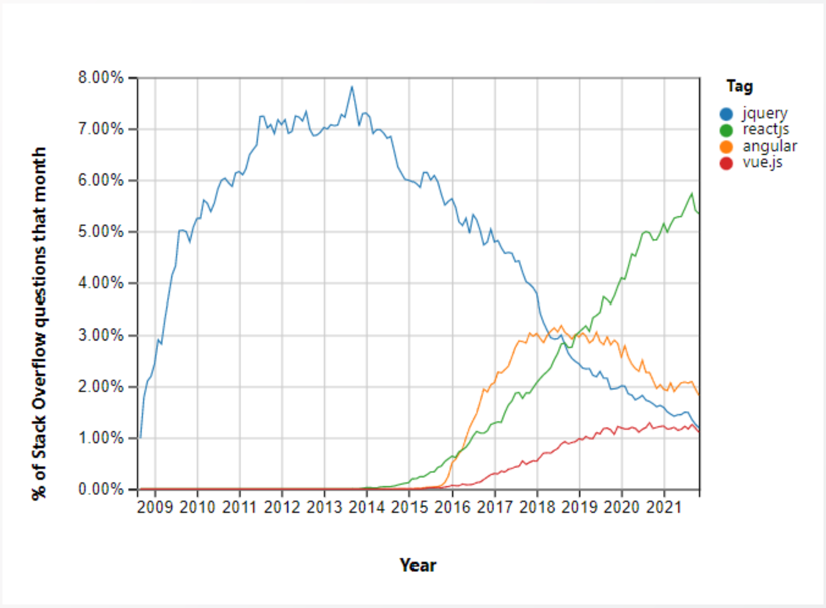
Introduction

ReactJS is a JavaScript library released by Facebook in 2013 for building modern applications. ReactJS has a diverse ecosystem with excellent libraries for literally everything you might need. However, it can be overwhelming to make so many choices to pick up the perfect one. React applications give you the freedom to write code in any way you like, but that comes with a cost. Since there is no official architecture developers can follow, it often leads to messy, inconsistent, and complicated codebases.

This is an attempt to demonstrate a way of creating React applications using the best way possible in the ecosystem with a suitable project structure and scalability. This ebook is based on the experience of working with many different applications over 4+ years.

Why React?

React is very powerful in building modern applications and backed by a strong community, making React stand out against other frameworks and libraries. Let the number show you why you should go ahead with React.



Prerequisites to learn ReactJS

- HTML
- CSS
- JavaScript

Suppose you don't feel very confident with the above list. In that case, we recommend going through a specific tutorial to check your knowledge level and enable you to follow along with this guide without getting lost. Having fundamentals cleared before diving into React is essential to keep the ball rolling.

Hello World

> 💡 It's a golden rule to start learning any programming language with Hello World.

PC Prerequisites

✅ NodeJS

✅ NPM or Yarn

Once you are done with the above checklist ^^. Open your favorite terminal and try below command.

```
npx create-react-app hello-world
```

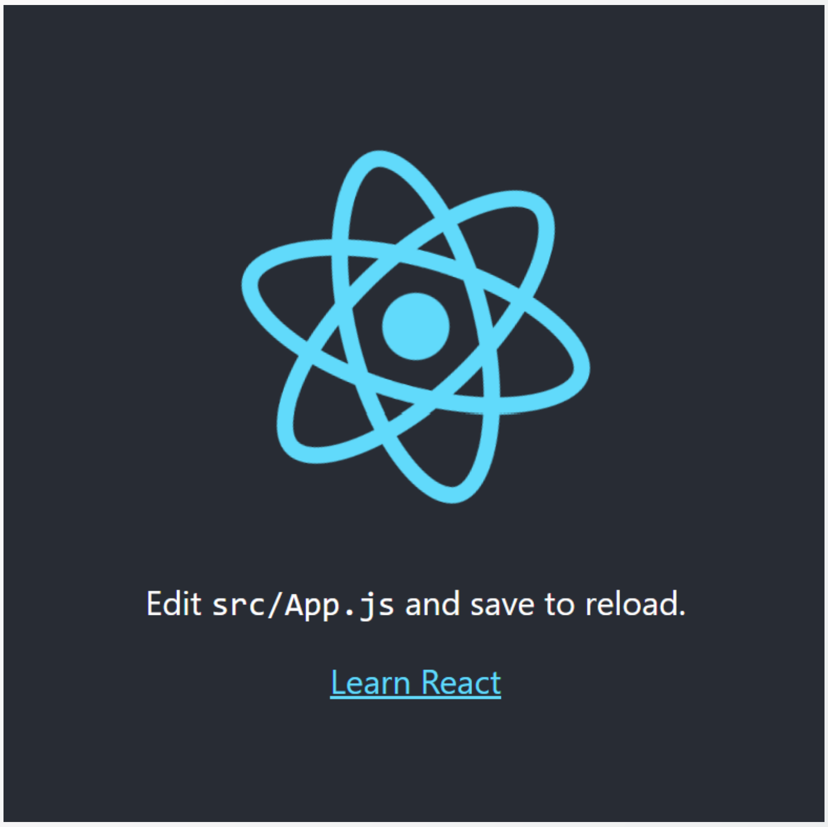
```
cd hello-world
```

```
npm start
```

This might take a time based on your PC configuration⏳.

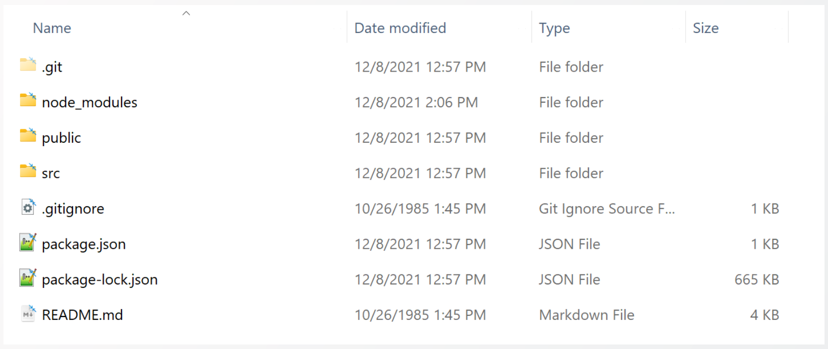
That's it.

You will see a local server running your React Application. It will open your default browser with URL: http://localhost:3000/



Now let's see what had happened when we ran **create-react-app** command.

Open folder in which you ran the command. You will see bunch of files.



Name	Date modified	Type	Size
.git	12/8/2021 12:57 PM	File folder	
node_modules	12/8/2021 2:06 PM	File folder	
public	12/8/2021 12:57 PM	File folder	
src	12/8/2021 12:57 PM	File folder	
.gitignore	10/26/1985 1:45 PM	Git Ignore Source F...	1 KB
package.json	12/8/2021 12:57 PM	JSON File	1 KB
package-lock.json	12/8/2021 12:57 PM	JSON File	665 KB
README.md	10/26/1985 1:45 PM	Markdown File	4 KB

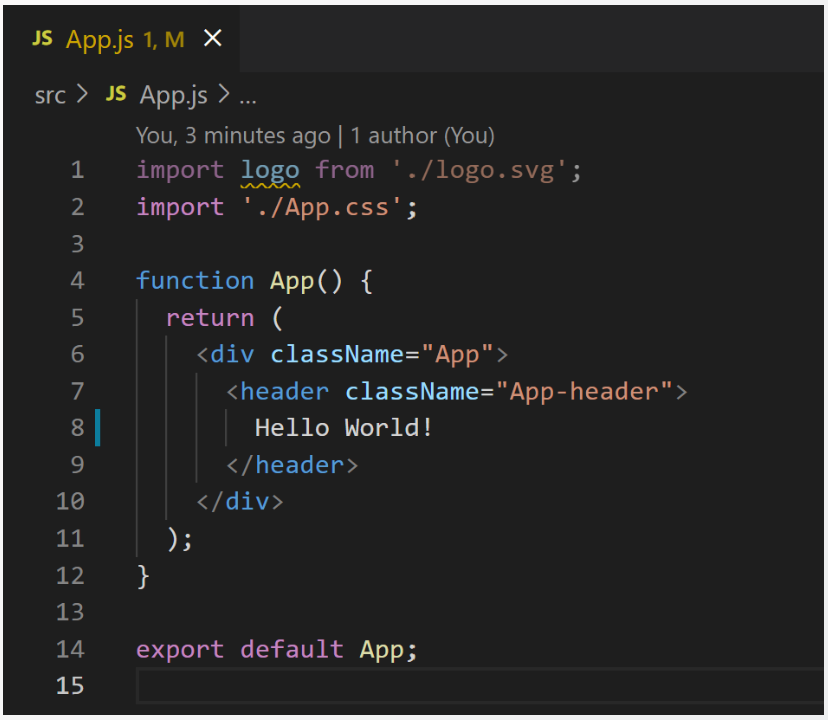
- .git & .gitignore -> Version control internal files.
- node_modules -> 3rd party library folders including ReactJS files.
- public & src -> These are the main folders that we are going to work upon.
- package.json -> It contains the all the meta information regarding project name,

libraries to run the project, scripts etc.

> npx stands for **Node Package Execute** and it comes along with the npm (version > 5.2.0). Generally NPX is used when you want to use the package only once.

We will go deeper in the upcoming chapters. Now let's try to edit the App.js file in the src directory. Remove everything inside `<header>` and add **Hello World!** Notice that our code is automatically reflected on the local server on saving a new code.

Great Job! 🚀



```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        Hello World!
      </header>
    </div>
  );
}

export default App;
```

Online Playgrounds

If you're interested in playing around with React without installing anything on your PC, you can use an online code playground.

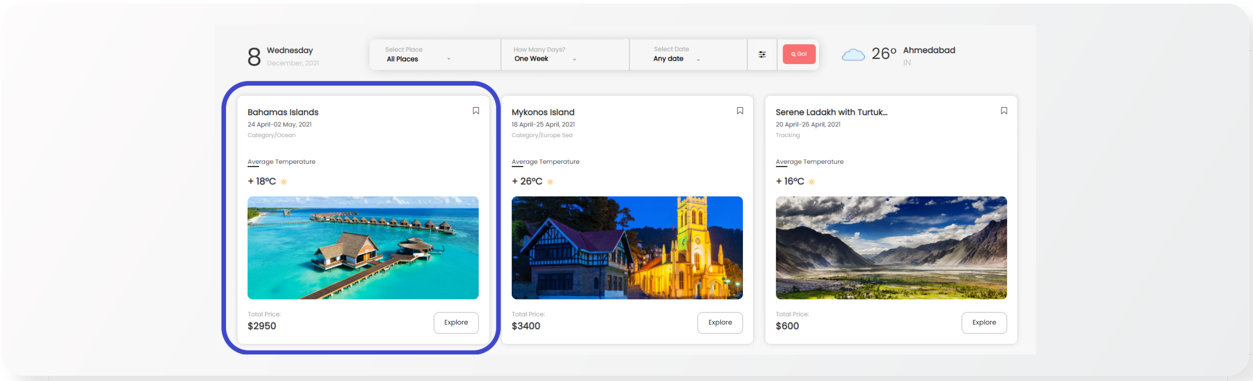
Try a Hello World template on CodePen, CodeSandbox

Core concepts

Components

Imagine we have a single webpage. Now if we need to divide that page into smaller independent UI by keeping reusability in mind. That smaller portion (highlighted with a blue border) is a component for us.

> Thinking in Components is how you work with React.



Components can be broken down into N levels based on a UI. In the above screenshot, a card is repeated. So we can consider `<Card>` as one component.

You might think that all the cards have different information like name, cover photo, price, etc. So should we consider this as a reusable component?

Yes, we can create a card that renders the data based on the data passed (called *props*).

Also, notice that in a card, we have a `<Button>` to create it as an independent component and place it inside `<Card>`.

> Note: Custom react components always starts with the capital letter.

Components can be divided into two main parts.

1. Class Components
2. Functional Components

Class Components

As the name suggests, class based components are child classes for the component class of React.

All the class components have access to the state that stores the current behavior and appearance of the component. Whenever we want to modify the state, we call the built-in **setState** function, which will automatically update the DOM.

The state can hold any values like a number, boolean, object, array of an object.

Each of the class components also requires a `render()` method. This method is responsible for returning HTML.

Here is the simple example:

```
class Message extends React.Component {
  render() {
    return <h2>Hello World!</h2>;
  }
}
```

The class component has the `constructor()` function called when the component is initialized. constructor method has `super()` function. Super refers to the parent class constructor. Super can be used to call the parent class variables and methods. `super()` can be used to call parent class constructors only.

All properties should be kept in object call `state` ,

Here a sample example:

```
class Message extends React.Component {
  constructor() {
    super();
    this.state = {
      message: "How are you?"
    };
  }
  render() {
    return <h2>Hello World! {this.state.message}</h2>;
  }
}
```

As show in the above example, the value which is stored in the state is consumed during the rendering of the HTML in the `render()` function.

setState is an async operation and React tries to combine the parallel calls for performance improvements. If you want to perform something after the value is reflected inside React State, you should use callback method of this.

```
this.setState({ someFlag: false },
  ()=> {
  // some operation
});
```

Functional Components

A functional component is a function that takes props and returns the JSX. They do not have access to the state or lifecycle methods. However, React hooks' introduction in the 16.8 version makes it possible to manage a state in the functional component.

Functional components are easier to read, debug and test. They are lightweight when compared to class-based components.

They are sometimes referred to dumb or stateless components as they accept the data and display it in the UI.

There is no render method in the case of the functional component.

Here is simple example of Functional component:

```
import React from "react";

const Person = props => (
  <div>
    <h1>Hello, {props.name}</h1>
  </div>
);

export default Person;
```

Class vs Functional

Class Component	Functional Component
We have a render function used to return React element	There is no render function
Class component has both options, can use the props and state	Functional components only accepts props as an argument
It is also called as Stateful component	It is sometime called as stateless components (With React Hook it is now possible to maintain state in React)
`this` keyboard can be used	`this` keyboard cannot be used
More feature rich	Simple function
We can't use Hooks with class component	We can use Hooks with functional components

State vs Props

State are commonly used to manage the internal environment of the components means that data is changed inside the components. It cannot be accessed or modified outside of the component. In order words, it is an equivalent local variable in a function.

A component commonly uses props to get data from an external environment, i.e., in most cases is another component. They make components reusable by giving components the ability to receive data from their parent component. In order words, it is equivalent to function parameters.

What can props and state do?

Description	State	Props
Can get initial value from parent Component?	Yes	Yes
Can change inside Component?	Yes	No
Can be changed by parent Component?	No	Yes
Can change in child Components?	No	Yes
Are mutable?	Yes	No
Can set initial value for child Components?	Yes	Yes

Does every component need to have a state?

No, the state is optional. It increases the complexity and reduces the predictability.

What are the component types?

1. **Stateless Component**
    a. It has only Props and No state.
    b. Here there is nothing much going on inside the `render()` function. It directly displays the data based on whatever props are received. Thus it becomes easy to test and use them.
2. **Stateful Component**
    a. It has both state and props.
    b. They are used when components have to retain some state. An excellent example of this could be client-server communication dealing with XHR and Web Sockets.

Class Component Lifecycle

The three phases are:

1. **Mounting**
2. **Updating**
3. **Unmounting**

Let's look at them one by one and understand them better. Here is a quick diagram that will help you remember things better when working with class-based components.

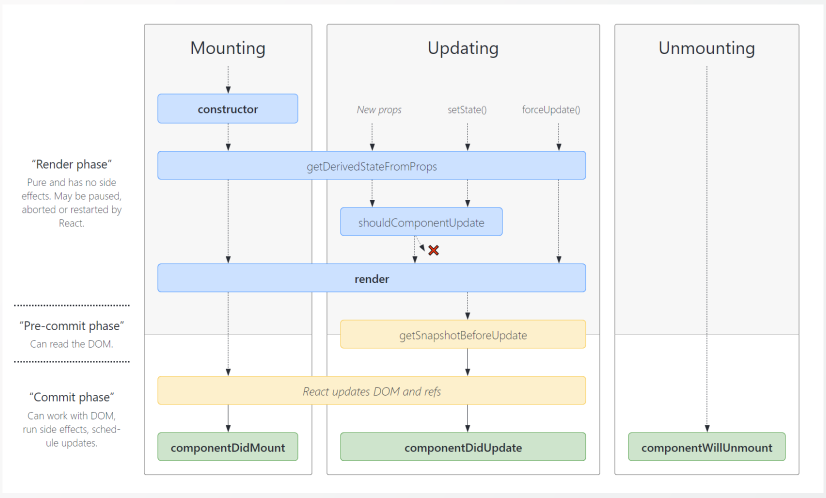


Diagram Source

Mounting

Mounting is nothing but the time when elements are added to the DOM. Four methods get called when mounting of component happens.

1. constructor()

This is the first method that gets called before anything else when the component is initiated. This is where we set up the initial state, which we will be used within the component.

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 allow the component to inherit methods from its parent (`React.Component`).

Why is the constructor needed?

1. Initializing the local state that will be used inside the component by assigning the object to `this.state` .
2. Bind event handler methods that are used within the component.

Why is super needed?

```
class Input extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isReadOnly: true };
  }
  // ...
}
```

1. Super refers to the parent class constructor. In the above example, it points to `React.Component` .
2. You can't use this in a constructor until *after* you've called the parent constructor.

```
class Input extends React.Component {
  constructor(props) {
    // 🔴 Can’t use `this` yet
    super(props);
    // ✅ Now it’s okay to use this
    this.state = { isReadOnly: true };
  }
  // ...
}
```

If we want to use `this` keyword in the constructor, we need to have super inside the constructor because ES6 class constructors must call super if they are subclasses.

We call `super(props)` inside the constructor if we have to use `this.props` , for example:

```
class Input extends React.component{
  constructor(props){
    super(props);
    console.log(this.props); // prints out whatever is inside props
  }
}
```

When to use it?

1. When you want to use `this` keyword in the component
2. Declare the initial state
3. When you are using class components

2. getDerivedStateFromProps()

This is a static method called just before `render()` method in both the mounting and updating phase in React. It should return an object to update the state or `null` to update nothing.

When to use it?

1. state of a component depends on changes of props

3. render()

React renders HTML to the web page by using a function called `render()`. This method is required and will always be called.

```
class App extends React.component {
    render(){
        return <div>Hello { this.props.world }</div>;
    }
}
```

In `render()` method we can read props and state and return our JSX code to the root component of our app.

4. componentDidMount()

This method gets called after the component is rendered. It allows us to execute the React code when the component is already placed in the DOM.

```
class Message extends React.Component {
  constructor(props) {
    super(props);
    this.state = {text: "Hello World"};
  }

  componentDidMount() {
    setTimeout(() => {
      this.setState({text: "I'm changed after 5 second"})
    }, 5000)
  }

  render() {
  const { text } = this.state;
    return (
      <h1>Here is the message {text}</h1>
    );
  }
}

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

When to use it?

1. If you want to load the data from the remote endpoint this is the best place to do that.

Updating

Once the mounting is done, the next phase in the lifecycle is when a component is getting updated.

There are 2 main reasons why the component gets updated

1. There is a change in the state value of the component

2. There is a change in the props of the component

As shown in the above lifecycle diagram, five built-in methods get called. The methods shown below are optional apart from the required render() method.

1. getDerivedStateFromProps()

This is the same method that gets called during a component's mounting. This is the first method that gets called when a component is updated.

2. shouldComponentUpdate()

This method gives you the freedom to have control over the rendering. Before the component re-renders itself, it gives you control to stop it.

You can specify whether React should continue rendering the DOM or ignore it by returning a Boolean value.

You can also set some conditions as this method gets called before the DOM update.

`shouldComponentUpdate()` is called every time:

1. The props are updated by every re-render of the parent component. This includes all scenario's where re-render takes place with exactly the same prop values.
2. The state is updated by a call to `setState()` (the only way allowed by react). This includes all scenario's where the value of state is exactly the same like the previous state.

```
shouldComponentUpdate(nextProps, nextState) {
  console.log(nextProps, nextState);
  console.log(this.props, this.state);

  // Check some conditions and decide
  // True => Component will render, DOM changes
  // False => Component will not render, DOM doesn't change
  return false;
}
```

3. render()

This method is a called every time a component is updated as it has to re-render the HTML to the DOM with all the new changes that has happened.

4. getSnapshotBeforeUpdate()

It let's you have the access to the state and props before the update. Even after the update, you can check what the values were *before* the update.

Note:
If you are using this method then you need to include `componentDidUpdate()` method else it will throw error.

```
getSnapshotBeforeUpdate() {
  return {foo: 1};
}

componentDidUpdate(prevProps, prevState, snapshot) {
  console.log(snapshot) // { foo: 1 }
}
```

The snapshot variable within the `componentDidUpdate` method can use the object that was returned in `getSnapshotBeforeUpdate` .

When to use it?

1. Capture some information about the DOM before it has changed

5. componentDidUpdate()

As the name suggests, this method is called immediately after the component is updated in the DOM. It doesn't get called in the initial render.

Note:
If you forget to compare the current props to previous props, it will result in unnecessary infinite re-rendering.

```
componentDidUpdate(prevProps) {
  // Typical usage (don't forget to compare props):
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

When to use it?

1. We need to call an external API because the previous state and the current state have changed.
2. If you work with the DOM nodes, get the component's position and dimensions or initialize the animation.

Unmounting

This is the last phase of the lifecycle, triggered when a component is removed from the DOM. There is only one built-in method in this lifecycle, as shown in the above lifecycle diagram.

1. componentWillUnmount()

This method is called when the component is about to be removed from the DOM or is destroyed.

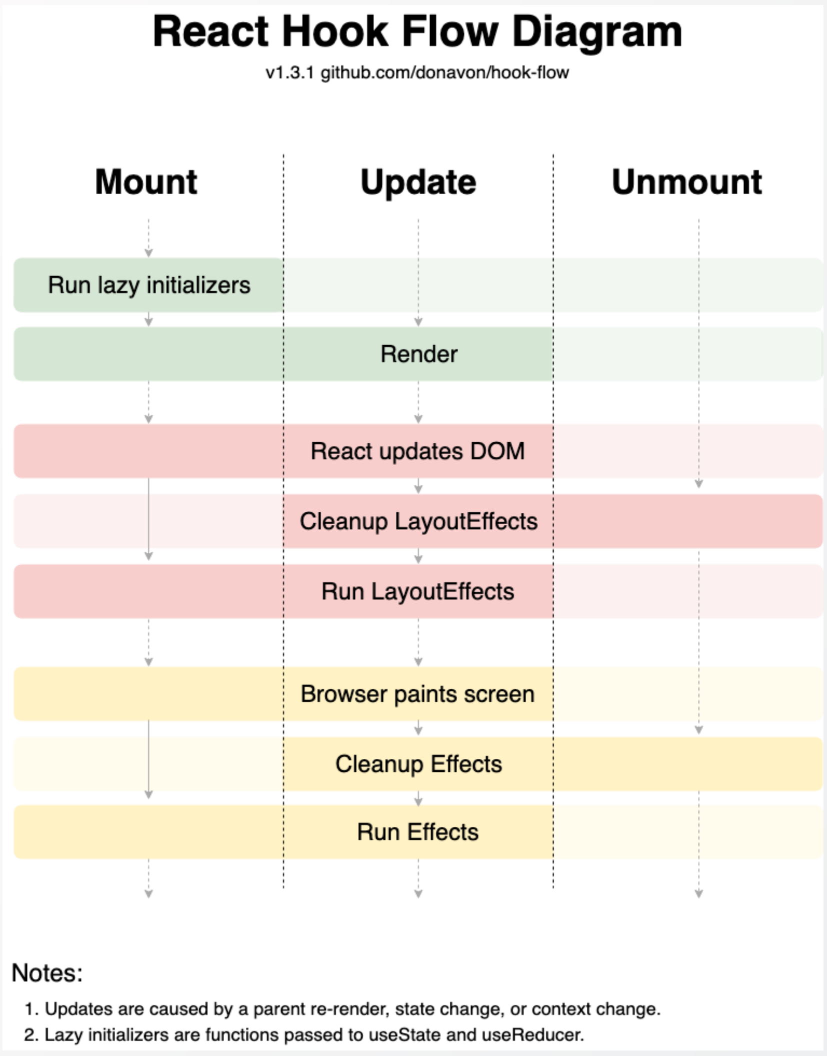
```
componentDidMount() {
   document.addEventListener("click", this.closeMenu);
}

componentWillUnmount() {
    // clean up happens here..
    document.removeEventListener("click", this.closeMenu);
}
```

When to use it?

1. It is mainly used to clean up the timers, cancel the network requests, or clean up any subscriptions or listeners.

   Please **don't** use it to set the state, as it will never update the DOM when the component itself is unmounted.

Functional Component Lifecycle



Notes:
1. Updates are caused by a parent re-render, state change, or context change.
2. Lazy initializers are functions passed to useState and useReducer.

Diagram Source

Hooks

With the release of React 16.8, Hooks where introduced. It's an ergonomic way to build components where stateful logic can be used without changing the component hierarchy.

There are ten different types of hooks. Let's go over them one by one.

1. **useState**

   useState is **a Hook that allows you to have state variables in functional components**.

   You pass the initial state to this function, and it returns a variable with the current state value (not necessarily the initial state) and another function to update this value.

   This is the most used Hook. The purpose of this Hook is to handle the reactive data. This is similar to a set class-based component state.

   When the count value is updated, UI is re-rendered. To update the count value, we use the setCount method.

   ```
   const [count, setCount] = React.useState(0);
   ```

2. **useEffect**

   The useEffect Hook allows **you to perform side effects in your components**.

   It can be used to execute actions when the component mounts or a particular prop or state is updated for the component and execute code when the component is about to unmount.

   useEffect executes a code that needs to be called during the lifecycle of the component instead of any specific DOM events.

```
import React, { useState, useEffect } from "react";

function Example() {
   const [count, setCount] = useState(0);

  useEffect(() => {
    console.log(`Equivalent of componentDidMount()`);
    // Things that needs to be cleanup should be put in the return
statement
    return () => {
      console.log(`Equivalent of componentWillUnmount()`);
    };
  }, []);

  // equivalent of componentDidUpdate()
  useEffect(() => {
    console.log(`You rendered ${count} times`);
  }, [count]);

  const handleButtonClick = () => {
    setCount(count + 1);
    console.log(`You clicked ${count + 1} times`);
  };

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Button 1</button>
      <button onClick={handleButtonClick}>Button 2</button>
    </div>
  );
}
```

`handleButtonClick` fires on `Button 1` click, while `useEffect` fires on every state change (according to the dependency array).

In the next example (Button 2), you notice that `useEffect` will log on every button click (`Button 1/2`), and `handleButtonClick` will log only on `Button 2` click.

3. **useContext**

   Primary use case of this is to pass the data throughout your app without needing to pass the prop at each level down the tree.

   It is a mechanism to share data that can be considered global data that all the components could share down the tree. It could be used to hold information like theme, authenticated user details.

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

// Create a Context
const SecretContext = React.createContext();
// It returns an object with 2 values:
// { Provider, Consumer }

function App() {
  // All the things which is wrapped inside the provide
  // ( children and grandchildren ) can access the value
  return (
    <SecretContext.Provider value={455878}>
      <div>
        <Display />
      </div>
    </SecretContext.Provider>
  );
}

function Display() {
  // Consume is used to access the value from context
  // No props are used but still data can be accessed
  return (
    <SecretContext.Consumer>
      {value => <div>The Secert code is {value}.</div>}
    </SecretContext.Consumer>
  );
}

ReactDOM.render(<App />, document.querySelector("#root"));
```

4. **useRef**

   In the functional component, `useRef` allows directly creating a reference to the DOM element.

   It is a mutable ref with the object having a property called `.current`.

   Two things to remember while using this hook

   a. The value of the reference is persisted, meaning it will stay the same
   b. When you update the reference, it will not trigger a re-rendering

Let's take a look at the example

```
import { useRef, useEffect } from 'react';
function InputFocus() {
  // inputRef is initialized
  const inputRef = useRef();
  useEffect(() => {
 // on page load,
 // with the use of current DOM is accessed and input is focused
    inputRef.current.focus();
  }, []);

  return (
    <input
      ref={inputRef}
      type="text"
    />
  );
}
```

5. **useReducer**

`useReducer` is used for complex state manipulations and state transitions that involve multiple sub-values or when the state depends on the previous one.

It helps in performance optimization instead of passing the callbacks. You pass a dispatch event.

It accepts two things

1. a reducer function which usually has two parameters state and action
2. initial state

```
const initialState = {count: 0};

  function reducer(state, action) {
    switch (action.type) {
      case 'increment':
        return {count: state.count + 1};
      case 'decrement':
        return {count: state.count - 1};
      default:
        throw new Error();
    }
  }

  function Counter() {

    const [state, dispatch] = useReducer(reducer, initialState);
    return (
      <div>
        Count: {state.count}
        <button onClick={() => dispatch({type: 'decrement'})}> -
</button>
        <button onClick={() => dispatch({type: 'increment'})}> +
</button>
      </div>
    );
  }
```

6. **useMemo**

React has a built-in hook called `useMemo` that allows you to memoize expensive functions so that you can avoid calling them on every render. `useMemo` does the memoization of value so it does not need to re-calculate itself for every re-render.

It tries to cache the result, for example if we have a multiplication function where two numbers are passed and the result is returned so for example, If I pass 4 and 5 it will return 20 doing the calculation, but next time if I pass the same input it will not do calculation since it was previously calculated. It will directly return 20. Next time meaning back to back i.e 4 and 5 are passed twice.

Similar if in React, if we use Memo the state and props do not change the component and components does not re-render.

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

  export function CalculateFactorial() {
    const [number, setNumber] = useState(1);
    const [inc, setInc] = useState(0);

  // using Memo
    const factorial = useMemo(() => factorialOf(number), [number]);

    const onChange = event => {
       setNumber(Number(event.target.value));
    };

    const onClick = () => setInc(i => i + 1);

  return (
      <div>
        Factorial of
        <input type="number" value={number} onChange={onChange} />
        is {factorial}
        <button onClick={onClick}>Re-render it</button>
      </div>
    );
  }

  function factorialOf(n) {
    console.log('factorialOf(n) function is called!');
    return n <= 0 ? 1 : n * factorialOf(n - 1);
  }
```

Let's assume the case if useMemo was not used in the above code snippet then, Every time you change the input value, the factorial is calculated factorialOf(n) and `factorialOf(n) function is called!` is logged to console. Apart from it, rendering would happen each time you click on *Re-render.*

But with Memo, every time you change the value of the number, `'factorialOf(n) function is called!` is logged to console. That's expected. However, if you click *Re-render* button, `factorialOf(n) function is called!` isn't logged to console because `useMemo(() => factorialOf(number), [number])` returns the memoized factorial calculation.

7. useCallback

The `useMemo` and `useCallback` Hooks are similar. The main difference is that useMemo returns a memoized value, and `useCallback` returns a memoized function.
We can use the `useCallback` hook to prevent the function from being recreated unless necessary.

```
import React, { useState, useCallback } from 'react'
var funccount = new Set();
const App = () => {

  const [count, setCount] = useState(0)
  const [number, setNumber] = useState(0)

  const incrementCounter = useCallback(() => {
    setCount(count + 1)
  }, [count])

  const decrementCounter = useCallback(() => {
    setCount(count - 1)
  }, [count])

  const incrementNumber = useCallback(() => {
    setNumber(number + 1)
  }, [number])

  funccount.add(incrementCounter);
  funccount.add(decrementCounter);
  funccount.add(incrementNumber);

  alert(funccount.size);

    return (
      <div>
        Count: {count}
        <button onClick={incrementCounter}>
           Increase counter
        </button>
        <button onClick={decrementCounter}>
           Decrease Counter
        </button>
        <button onClick={incrementNumber}>
           increase number
        </button>
      </div>
    )
  }

export default App;
```

In the above example, if the useCallback wasn't there, all three functions are recreated again, and the alert increases by three at a time.

But after using useCallback, When we change the state count, then two functions will be re-instantiated, so the set size will be increased by 2, and when we update the state ‘name’, then only one function will be re-instantiated. The size of the set will increase by only one.

8. useImperativeHandle

This hook is mainly used to pass values and functions from a child component back to the parent component using a **ref**.

The parent component can then pass it to another Child or use it within that component.

You can only pass a ref as a prop to the child, if it wraps a component in forwardRef.

```jsx
import React, { useRef, useImperativeHandle, forwardRef } from 'react';

function Input(props, ref) {
  // create useRef
  const btn = useRef();

  useImperativeHandle(ref, () => ({
    focus: () => {
      console.log('Input is in focus');
    },
  }));

  return <input ref={btn} {...props} placeholder="Enter name" />;
}

export default forwardRef(Input);
```

Input.js

```jsx
import React, { useRef } from 'react';
import Input from './Input';

const App = () => {

  const inputRef = useRef(null);

  return (
    <div>
      <Input onFocus={() => inputRef.current.focus()}
      ref={inputRef} />
    </div>
  );
};

export default App;
```

App.js

9. useLayoutEffect**

This hook performs immediately after React has performed all the DOM mutations. Thus this is useful when you need to make DOM Measurements like getting the scroll position or other styles of an element.

This is used to read layout from the DOM and synchronously re-render. Updates scheduled inside `useLayoutEffect` will be flushed synchronously before the browser has a chance to paint.

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

const App = () => {
  const [value, setValue] = useState('Home');

  useLayoutEffect(() => {
    if (value === 'Home') {
     // Changing the state
      setValue('World');
    }
    console.log('UseLayoutEffect is called with the value of ', value);
  }, [value]);

  return <div>{value} is where you live!</div>;
};

export default App;
```

10. useDebugValue

useDebugValue is used to display a label for custom hooks in React DevTools. It takes any value that we want to display.

Whatever is passed inside the useDebugValue will be printed inside the React DevTools.

```
import React, { useDebugValue, useState } from "react"

export default function useExampleHook() {
  const [b, setB] = useState(false)

  // defining the log message
  useDebugValue("This code is executed")
  return [b, setB]
}

export default function App() {
  useExampleHook()
  return <div>Hello World</div>div>
}
```

Virtual DOM

DOM is nothing but a "Document Object Model." It represents the UI of your application. When there is a change in the state or props of your application, the DOM gets updated to represent the new changes.

Imagine if the DOM needs to be updated frequently, then it would result in the application's performance.

DOM is a tree structure data, But whenever there is change, the updated elements and its children have to be re-rendered, to update the application UI.

The more components = The more expensive is the updating of the UI.

What change does virtual DOM brings to the table

The virtual DOM is a virtual representation of the DOM. Whenever the state of the application changes, the virtual DOM is the one that's getting updated instead of the real DOM.

Virtual DOM is a mimic of the real DOM, and it compares the difference with the previous virtual DOM tree. In short, it helps to calculate the best possible method to make the changes in the real DOM.

This reduces the performance cost of updating real DOM as instead of re-rendering a complete DOM, it only tries to update that node where the new changes have occurred.

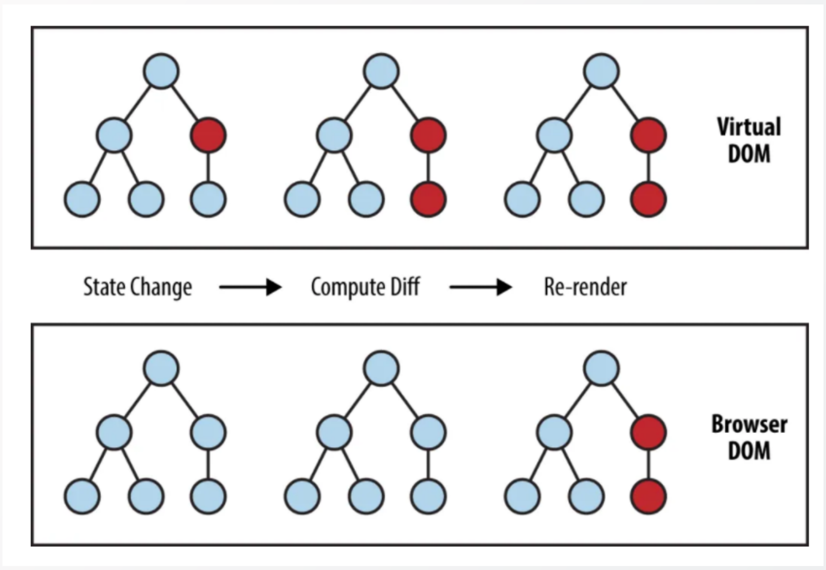


Diagram Source

In the above diagram, red circles are the nodes that are changed. These nodes are the UI elements that have had their states updated.

The difference between the previous version of the virtual DOM tree and the current virtual DOM is calculated, and after that whole parent, subtree gets re-rendered to give the updated UI. This updated tree is then batch updated to real DOM. All changes at once for better performance.

Few things to remember

1. Frequent DOM manipulations are expensive and often performance-heavy.
2. Virtual DOM is a virtual representation of the real DOM.
3. The Virtual DOM sends a batch update to the real DOM to update the UI.
4. Virtual DOM technical is the reason for the boost in React performance.
5. React internal uses a lightweight comparison algorithm to detect the difference.

Higher Order Component

As developers, we always prefer to reuse code for better maintenance and readability. React allows the facility to reuse the component with the Higher-Order Component (HOC) technique. It's a compositional pattern.

In short, a higher-order component takes a component as a parameter and returns a new component.

Many libraries provide the feature with HOC. Example: React-sortable-hoc

`const SuperComponent = CustomHoc(SimpleComponent);`

Let's try to create one simple example to better understand the usage.

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

```
const SuperComponent = (props) => {
    const { component: Component } = props;
    return (
        <div className='super-awesome-class'>
        {/* Some extra styling wraping */}
            <Component {...prop} />
        </div>
        )
}
```

So, while using HOC,

```
// Some class & import statements
// ...
render() {
    return (
      <SuperComponent component={SimpleComponent} extraProps=
{SomeData}/>
    )
}
```

In the above example, we have just wrapped the component with extra `<Div>` with styles, but HOC can have many use cases.

Let's try one more example. Assume that our React App is an Admin site with multiple pages. And some of the admin pages are super secure and can only be accessed by Super Admin.

```
const App = () => {
  render() {
    return (
      <BrowserRouter>
        <Switch>
          <Route path="/normal-page" component={NormalPage} exact />
          <Route path="/super-secure" component={SuperSecurePage} />
        </Switch>
      </BrowserRouter>
  }
};

const SuperSecurePage = () => {
  <div>Content that only Super Admin can see</div>
};

const NormalPage = () => {
  <div>Normal page that other users can see</div>
};
```

Ok, so we have 2 pages. We have used react-router-dom to support multiple pages based on URL. We are going to create `<PrivateRoute>` with HOC which will wrap the normal `<Route>`.

```
// Import Modules
import React from 'react';
import { Route, Redirect } from 'react-router-dom';

const NotAuthorized = () => {
  <div>You are not authorized to access this page.</div>
}

const PrivateRoute = (props) => {
  const { component: Component } = props;
  // some function to fetch the user Token with permissions array.
  const token = GetToken();
  const checkIfSuperAdmin = token.permissions.includes('super-admin');

  return (
    <Route
      {...props}
      render={prop => (checkIfSuperAdmin ? <Component {...prop} /> :
<NotAuthorized />)}
    />
  );
};
```

Context

Earlier we learned about passing a data from parent to child components with use of props like below example.

```
Class Parent extends React.Component {
   ...
   ...

  render (){
    const data = {
      // some data
    }

    return (
      <div>
          <Child data={data} />
      </div>)
  }
}
```

Now imagine we have a nested tree with 10 layers of child components. And we need to pass the data from the parent to the last child. Also, intermediate children have nothing to do with the data.

```
Class Parent extends React.Component {
  ...
  ...

  render (){
    const data = {
      name: 'Dipen',
      // some more data
    }
    return (
      <div>
          <FirstChild data={data}>
              <SecondChild data={props.data}>
                  ...
                    <TenthChild data={props.data}>
                      {/* Actual use of data */}
                    </TenthChild>
                  ...
                </SecondChild>
          </FirstChild>
      </div>)
  }
}
```

Well, this is not perfect but it will work. Now imagine you have two more props to be passed, you have to update in each component.

ok, how about we use spread operator?

```
<FirstChild data={data}>
  <SecondChild {... props}>
    ...
    <TenthChild {... props}>
      {/* Actual use of data */}
    </TenthChild>
    ...
  </SecondChild>
</FirstChild>
```

Again this will work, but it will also pass all the unnecessary props from parent to child.

So, What next? Redux?

Why?

1. Context API is easy to use with a short learning curve.
2. It requires less code as there's no need for extra libraries, bundle sizes are reduced.
3. On the other hand, Redux requires adding more libraries to the application bundle. The syntax is complex and extensive, creating unnecessary work and complexity.

Context to the rescue

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

There are 3 easy steps to use context.

1. Create context using the `createContext` method.

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

2. Take your created context and wrap the context provider around your component tree and put any value you like on your context provider using the `value` prop.

   ```
   <DataContext.Provider value={data}>
       <FirstChild>
          ...
     </FirstChild>
   </DataContext.Provider>
   ```

3. Read that value within any component by using the context consumer

```
<FirstChild>
  ...
    <TenthChild>
    </TenthChild>
  ...
</FirstChild>

function TenthChild(){
  return (
    <DataContext.Consumer>
        {value => <h1>{value.name}</h1>}
        {/* prints: Dipen */}
    </DataContext.Consumer>
  )
}
```

- With React 16.8, we can directly use **useContext** hook to consume the data.

```
function TenthChild(){
    const value = React.useContext(DataContext);
    return (
      <h1>{value.name}</h1>}
        {/* prints: Dipen */}
    )
}
```

That's it, we can read the data directly in the `<TenthChild />` w/o doing anything on intermediate children. You can either use the consumer component directly or the useContext hook, depending on which pattern you prefer.

Also, if we update the data, any component which consumes that context will re-render.

> Note: Don't use the context right away. See if you can better organize your components to avoid prop drilling.

There can be multiple consumers, for a single provider as well. Providers can also be nested to override values deeper within the tree.

Context object also accepts displayName as a string property.

```
const DataContext = React.createContext(defaultValue);
DataContext.displayName = 'MyContextComponent'
```

This will appear as `MyContextComponent` in the DevTools.

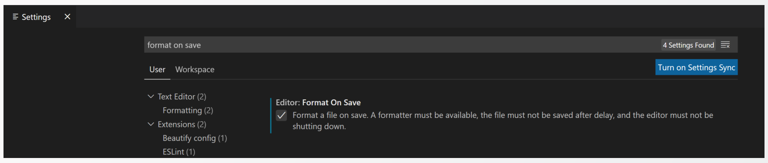
Editor Setup

A proper editor configuration can boost your speed up to 40% with cleaner code, better readability. Also, it can help you catch bugs. VS Code is most trending for development, and here are some tips for tuning your configuration.

> Note: ESLint is open source linter for JavaScript!

- Install ESLint
- Integrate ESLint in VSCode
- Prettier Extension

- Simple React Snippets Extension
- React Native Tools Extension
- Turn on Auto Save (Settings -> Search for *Format on save* -> Select Checkbox)



Editor: **Format On Save**

✓ Format a file on save. A formatter must be available, the file must not be saved after delay, and the editor must not be shutting down.

- React Developer Tools
  - Chrome or Brave
  - Firefox
  - Edge
- Last and the most powerful extension GitHub Copilot

> 💡 To know more about React Developer Tools

Styling

CSS Style

This is the common approach that we are used to it while developing HTML pages. Just create a CSS file and import it.

It is recommended to break styles into separate files based on components and common usage for maintainability.

app.css

```
.hello-world {
  color: '#333';
}
```

app.jsx

```
import './app.css';

function App() {
  return (
    <div className="hello-world">
      Hello World!
    </div>
  );
}

export default App;
```

You can also use *SCSS* if you want. Just run below command

`npm install --save-dev node-sass`

We can now rename our App.css to App.scss.

Notice that, while installation, we have used **--save-dev** flag. This is used to install dev dependencies packages. i.e., packages that we only need for development purposes are not necessary to run the code in production.

```
"devDependencies": {
  "node-sass": "^4.13.1",
}
```

To auto-generate the CSS file in the development, we will install npm-run-all package. This package allows us to do parallel operations to help us achieve hot reload. Let's update our *package.json* file as below.

```
"scripts": {
  "start": "react-scripts start",
  "dev": "run-p start watch:sass",
  "watch:sass": "npm run build:sass && npm run build:sass -- -w",
  "build:sass": "node-sass -r --output-style compressed src/App.scss -o
src/",
   // other scripts
 },
```

You can customize the command to watch for entire folder as well. Refer node-sass doc for more details.

Inline Style

Just like what we used to do in a pure HTML file by adding a style attribute to the element, we can apply the style to React component. Check the below example in which we have used font size.

```
import './App.css';

function App() {
 return (
    <div className="App">
     <header className="App-header" style={{fontSize: "30px"}}>
          Hello World!
     </header>
  </div>
 );
}

export default App;
```

But what if we need to apply many properties? If we put everything on a single line, it will kill the readability of the code. We can create a style object and pass it to the style attribute.

```
import './App.css';

function App() {
  const headerStyle = {
    fontSize: '30px',
    color: '#F05050',
    // Many more style properties
  }

  return (
    <div className="App">
      <header className="App-header" style={headerStyle}>
        Hello World!
      </header>
    </div>
  );
}

export default App;
```

> 💡 CSS classes are generally better for performance than inline styles.

Styled Component

CSS-in-JS is becoming a popular choice as it offers a better API for developers to work with. Conclusion: There are many alternative frameworks like Radium and Ant Design, but I would recommend going with Styled components.

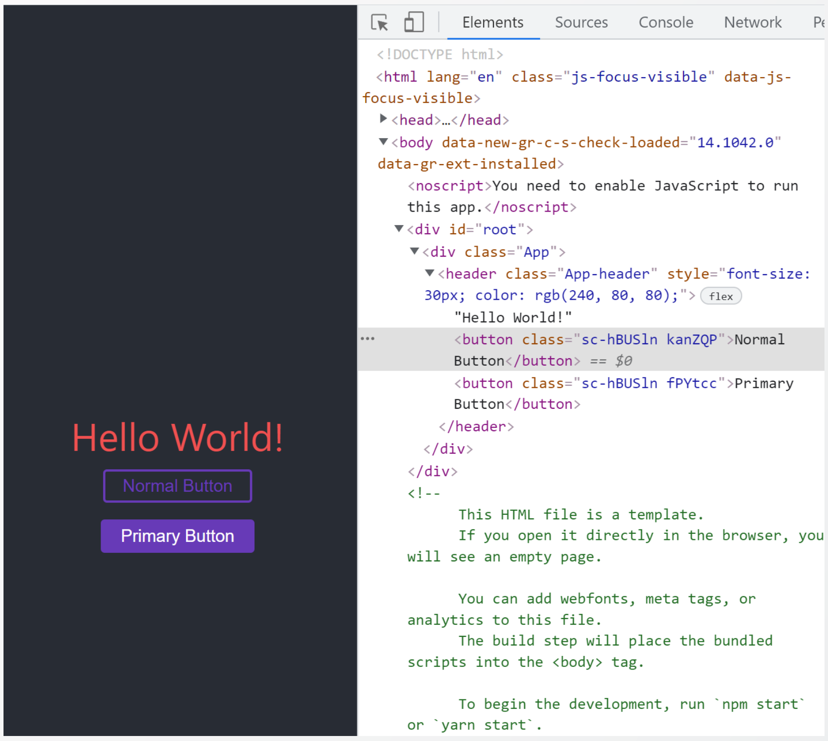
Let's start by installing package.

```
npm i styled-components --save
```

Styled components lets you create reusable component which you can tweak easily by passing props.

```
import './App.css';
import styled, { css } from 'styled-components'

function App() {
  const headerStyle = {
    fontSize: '30px',
    color: '#F05050',
    // Many more style properties
  }

  const Button = styled.button`
    background: transparent;
    border-radius: 3px;
    cursor: pointer;
    border: 2px solid #673ab7;
    color: #673ab7;
    margin: 0.5em 1em;
    padding: 0.25em 1em;

    ${props => props.primary && css`
      background: #673ab7;
      color: white;
    `}
  `;

  return (
    <div className="App">
      <header className="App-header" style={headerStyle}>
        Hello World!
        <Button>Normal Button</Button>
        <Button primary>Primary Button</Button>
      </header>
    </div>
  );
}

export default App;
```

In the above example, we created a simple button that can be customized by passing *primary* props. Also, styled components create a unique className. So we don't need to worry about the duplicate class name issue.



```
<!DOCTYPE html>
<html lang="en" class="js-focus-visible" data-js-
focus-visible>
 ▶<head>…</head>
 ▼<body data-new-gr-c-s-check-loaded="14.1042.0"
 data-gr-ext-installed>
    <noscript>You need to enable JavaScript to run
    this app.</noscript>
   ▼<div id="root">
     ▼<div class="App">
       ▼<header class="App-header" style="font-size:
        30px; color: rgb(240, 80, 80);"> flex
          "Hello World!"
          <button class="sc-hBUSln kanZQP">Normal
          Button</button> == $0
          <button class="sc-hBUSln fPYtcc">Primary
          Button</button>
        </header>
      </div>
    </div>
    <!--
          This HTML file is a template.
          If you open it directly in the browser, you
    will see an empty page.

          You can add webfonts, meta tags, or
    analytics to this file.
          The build step will place the bundled
    scripts into the <body> tag.

          To begin the development, run `npm start`
    or `yarn start`.
```

There are so many features available in the official doc And it's the best place to explore.

There are few things that you should consider while using a Styled components.

1. Avoid adding styled components in the render section. If your UI is huge and changing a lot, styled components will be recreated each time and will drain memory.
2. Static components (no dependency on props) are faster to load.
3. Nested style components can also impact on performance.

Error Handling

As a developer, we tend to overlook error handling, especially front-end development. In the ideal case, the application should handle all the possible errors. There are several ways we can handle errors in the ReactJS application.

Error Boundaries

A user interface shouldn't break in case of any errors. In React 16, a new concept is introduced to handle errors. 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 showing an error/broken page.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // You can also log the error to an error reporting service
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

This class component will work as an Error boundary. Now, we can directly wrap other components inside it, like below example:

```
<ErrorBoundary>
  <ComponentsWithErrorsToCatch />
</ErrorBoundary>
```

So if our *ComponentsWithErrorsToCatch* throws any error, *ErrorBoundary* will detect that error and display **Something went wrong** error message.

We can have multiple error boundaries based on our use cases. In case of multiple boundaries, the closest parent to the origin of the error will trigger. If an error occurs outside of an error boundary, the entire React component tree will be unmounted, crashing the application.

Error boundaries can only catch errors in the components below them in the tree. Error boundaries do not catch errors for:

1. Event handlers
2. Asynchronous code (e.g. setTimeout or requestAnimationFrame callbacks)
3. Server-side rendering

Try / Catch

We can use regular try/catch blocks when error boundaries fails.

```
try {
  await asyncFunction();
} catch (error) {
  // ...
}
```

react-error-boundary

Instead of handling errors in two different ways, we can use react-error-boundary package. It provides ready to use boundary class along with the **useErrorHandler** hook, to handle event handles & async operations. You can find the sample on the official docs.

Error Logging

We can always log in using console.log() or console.error(). It can help us troubleshoot the development errors. But what about the errors that occur to the end user's browser. How to track and handle those?

There are 3rd party libraries to handle those. I would recommend going with Sentry, Bugsnag as it's free for limited usage. And provides error hooks for Slack & Discord.

Deploy to the public use

As we are working with ***create-react-app***, we can create a deployment files ready just by running a below command.

`npm run build`

It will create a **build** folder with all the optimization possible. Now let's try to run this locally.

`npm install -g serve`
`serve -s build`

Looks good? But wait, if you see in the Network tab, you can see all the code along with the minification of the files. We can remove it by adding **GENERATE_SOURCEMAP=false** into our *.env* file.

Next we need to pick a service in which we would like to host our website. There are many free hosting provider with steps to deploy it.

Firebase

Netlify

Github Pages

If youhave multiple pages in your application and you would like to use Github pages, you need to do little bit tweaking for it.

- Add new entry into *package.json* file with

  `"homepage": "{http:// your-github-username.github.io}"`
- If you are using separate repo other than your user name

  `"homepage": "{http:// your-github-username.github.io/your-other-repo}"`
- Also, open *public/index.html* file and add the below snippet

```
(function (location) {
  if (location.search[1] === '/') {
    var decoded = location.search.slice(1).split('&').map(function (s)
{
      return s.replace(/~and~/g, '&')
    }).join('?');
    window.history.replaceState(null, null,
      location.pathname.slice(0, -1) + decoded + location.hash
    );
  }
}(window.location))
```

Performance Optimization

We have ReactJS website ready, and we need to boost its performance. Let's identify the key area in which we should plan to improve it.

Build size

The lesser the file size, the faster it loads. When we run npm run build, it creates production-ready files. If you check the logs, there should be a log about file size with gzip compression.

Now, let's try Source Map Explorer plugin. This will generate a static HTML/JSON report of which module/components are taking how much size. We can identify the package and use alternative lite packages as well.

While importing components, import only requires a module (2nd approach) instead of an entire package for lesser build size.

```
import { Button } from 'react-bootstrap';
```

OR

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

> Note: Also use CDN for faster page loading.

Code-Splitting

In case of enterprise applications, we might have lots of packages & many routes. Each route/components might be using different packages. So on the page's initial load, we don't need the entire bundled JS file during the initial load.

With Code-splitting the app, we can **lazy-load** just the things that are currently needed by the user, which can dramatically improve your app's performance.

import()

Dynamic import allow us to load modules only when it's needed. In below example, *special-module* is only imported when user perform some action and *handleClick* function is called.

```
handleClick = () => {
    import('./special-module')
      .then(({ module }) => {
        // Use module
      })
      .catch(err => {
        // Handle failure
      });
  };
```

React.lazy()

We can also do route-based code splitting into our app using *React.lazy()*.

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-
dom';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading ... </div>}>
      <Switch>
        <Route path="/home" component={Home}/>
        <Route path="/about" component={About}/>
      </Switch>
    </Suspense>
  </Router>
);
```

In above example we are loading components Home & About in the lazy mode. Notice that we are using **Suspense**. It's a mechanism for data fetching not the library to fetch the data. So in our example, while our component loads, it will show *Loading ...*

> Note: Suspense is a experimental features that aren't yet available in a stable release.

React.lazy currently only supports the default exports. If the module you want to import uses named exports then you can create an intermediate module that re-exports it as the default. This ensures that tree shaking keeps working and that you don't pull in all the unused components.

Service Worker

A service worker is a script that your browser runs in the background, separate from a web page. opening the door to extraordinary features like

1. Offline experience
2. Push Notifications
3. Caching local resources (JS, CSS, Images)

The service worker will intercept all the network requests. And it can return the locally cached files for static resources and API responses.

Web vitals

Since Google has started taking web vitals in account while indexing websites, you should keep an eye on web vitals scores from https://web.dev/measure/ & https://pagespeed.web.dev/

Code level optimization

React uses virtual DOM to minimize the DOM updation, but still, there is some way to improve the performance on your own.

useState Lazy Initialization With Function

There are times when we need to set the initial value of the state from some function that returns a value.

```
const initialState = someFunctionThatCalculatesValue(props);
const [count, setCount] = React.useState(initialState);
```

Since our function is in the body, every time a re-render happens, this function is getting called even if its value isn't required. We might only need it during the initial render. So we can do like this.

```
const getInitialState = (props) ⇒
someFunctionThatCalculatesValue(props);
const [count, setCount] = React.useState(getInitialState);
```

React.memo and useMemo

React.memo is built-in higher order component. If we have a component that return/render the same thing for same props, we can boost the performance of the component by wrapping it with *React.memo()*. React will memorize the result and skip the re-rendering of that component.

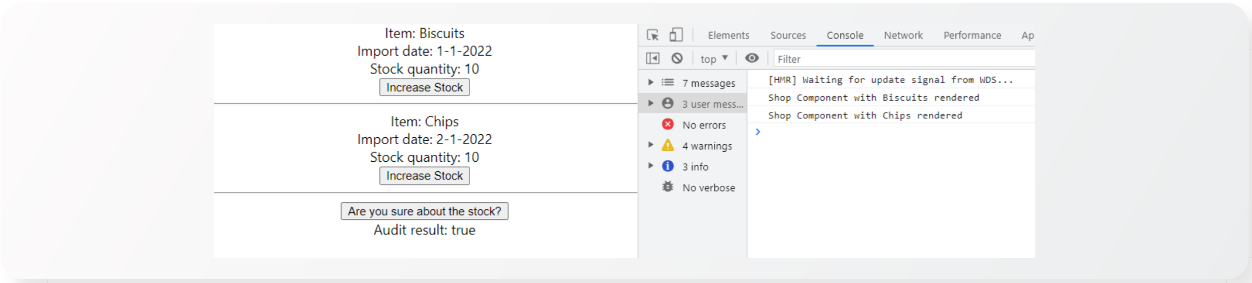
Let's try below example to better understand. Here's our App component:

```
import './App.css';
import React, { useState } from "react";
import Shop from './shop'

const SmartShop = React.memo(Shop);

function App() {
  const [report, setReport] = useState(true);
  const checkStatus = () => {
    setReport(!report);
  }

  return (
    <div className="App">
      <Shop stockItem={'Biscuits'} importDate={'1-1-2022'} />
      <hr />
      <SmartShop stockItem={'Chips'} importDate={'2-1-2022'} />
      <hr />

      <button onClick={checkStatus}>Are you sure about the stock?
</button>
      <div>Audit result: {report.toString()}</div>
    </div>
  );
}

export default App;
```

And here's our Shop component.

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

const Shop = ({ stockItem, importDate }) => {
    const [stockQty, setStockQty] = useState(10);
    console.log(`Shop Component with ${stockItem} rendered`);
    const updateStockQty = (stock) => {
        if (stock > 0) {
            setStockQty(stockQty+1);
        }
    }

    return (
        <div>
            <div>Item: {stockItem}</div>
            <div>Import date: {importDate}</div>
            <div>Stock quantity: {stockQty}</div>
            <button onClick={() =>updateStockQty(stockQty)}>Increase
stock</button>
        </div>
    );
}

export default Shop;
```

So, we have included component twice in while rendering. One regular & one with *React.memo()* Now if we see the console, both components are rendered.



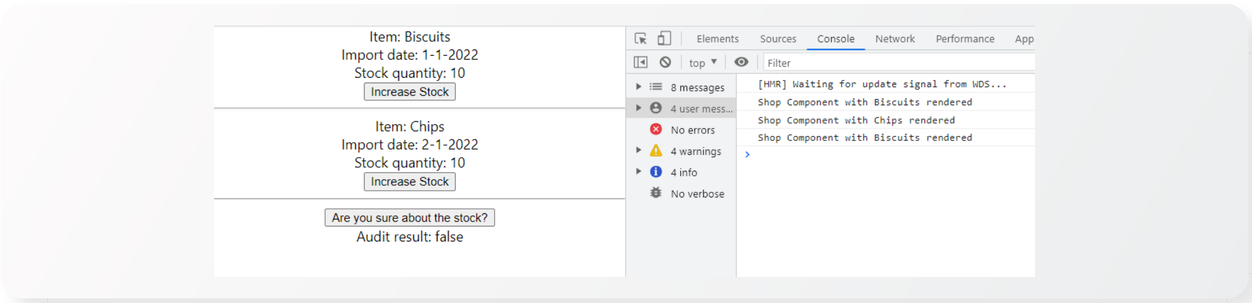
Item: Biscuits
Import date: 1-1-2022
Stock quantity: 10
Increase Stock

Item: Chips
Import date: 2-1-2022
Stock quantity: 10
Increase Stock

Are you sure about the stock?
Audit result: true

```
[HMR] Waiting for update signal from WDS...
Shop Component with Biscuits rendered
Shop Component with Chips rendered
```

Now if we click on `Are you sure..` button, log will look like below.



Item: Biscuits
Import date: 1-1-2022
Stock quantity: 10
Increase Stock

Item: Chips
Import date: 2-1-2022
Stock quantity: 10
Increase Stock

Are you sure about the stock?
Audit result: false

```
[HMR] Waiting for update signal from WDS...
Shop Component with Biscuits rendered
Shop Component with Chips rendered
Shop Component with Biscuits rendered
```

Shop with biscuits has rendered again while stocks with a chip haven't, as React.memo has memorized the result. Now, if we click on the `Increase Stock` button, it will re-render and increment the stock.

By default it will only shallowly compare complex objects in the props object. If we want to control over the comparison, we can do like this.

```
const areEqual = (prevProps, nextProps) => {
  /*
  return true if passing nextProps to render would return
  the same result as passing prevProps to render,
  otherwise return false
  */
}

const SmartShop = React.memo(Shop, areEqual);
```

> Note: React.memo will still re-render when state or context change. Make sure to use it wisely as this might lead to bugs as well.

We have gone through useMemo earlier.

Tweak shouldComponentUpdate method

We can always use *React.PureComponent()* instead of *React.Component()* while declaring a class component where for given props & state, our component render the same thing.

```
class Greeting extends React.PureComponent{
  //
}
```

So basically, *React.PureComponent* implement *shouldComponentUpdate()* with shallow comparison.

> Note: While using PureComponent, make sure all the children components are also pure. Otherwise it will leads to unusual bugs.

React.PureComponent is equivalent of *React.memo* for class component.

But what if we have nested objects in the props and we need to optimize the performance? We can use Deep Equal with *shouldComponentUpdate*.

React Window

When the list grows long, it's like poison is slowly entering your application and negatively impacting the performance.

It starts consuming a lot of memory in the browser. Since all the list items are in the DOM, there is a lag when you scroll the list.

React window works by only rendering part of a large data set, just enough to fill the viewport. Checkout this plugin here in the doc .

Custom Hooks

There are some use cases where we need to share logic between JS functions. Custom hooks allow us to extract component logic into reusable functions. The custom hook is one more way to share stateful logic between components. A custom Hook is a JavaScript function whose name starts with **use**.

Let's say we have some data to fetch, and once we get the data, we need to set it into our hook/state. And also, we have some authentications headers that we need to pass.

```
import { useState, useEffect } from "react";

const useFetch = (url) => {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch(url, {
        headers: customHeader //
    })
      .then((res) => res.json())
      .then((data) => setData(data));
  }, [url]);

  return [data];
};

export default useFetch;
```

We can also use React core hooks inside our custom hook too.

Now, we can use our custom hook useFetch to get the users & their todo list.

```
const User = () => {
  const [data] = useFetch("https://someurl-to-fetch/users");

  return (
      // ...
      );
}
```

```
const Todo = () => {
  const [data] = useFetch("https://someurl-to-fetch/todo");

  return (
      // ...
      );
}
```

Also, if we use useState inside our custom hook, and that hook is being used from two different components that will work out of the box as React isolate it's state.

> Note: There are some ready to use hooks available at useHooks

Folder Structure

There are many ways people are handling folder structures in ReactJS. The only goal of folder structure is to scale the application in the easiest and most maintainable way. Also, this is much easier to maintain than a flat folder structure with so many files.

```
src
|
+-- assets            # assets folder can contain all the static files
such as images, fonts, etc.
|
+-- components        # shared components used across the entire
application
|
+-- scene             # All the route pages
|
+-- services          # Layer to access API services
|
+-- styles            # All the stylesheet files
|
+-- test              # test utilities and mock server
|
+-- utils             # shared utility functions
|
+-- routes.jsx        # routes configuration
|
+-- index.jsx         # entry point
```

React Top NPM Packages to try

Here are 25 Hand-Picked React libraries you should try to save your time.

1. React Hot Toast (react-hot-toast.com)

   The Best Toast in Town. Smoking hot React notifications. Add beautiful notifications to your React App.

2. React Content Loader (skeletonreact.com)

   SVG-Powered component to easily create placeholder loadings (like Facebook's cards loading).

3. React Filepond (pqina.nl/filepond/)

   A new way to upload files

   - Multiple Input Formats
   - Image Optimization
   - Responsive
   - Async or Sync Uploading

4. React Notion X (github.com/NotionX/react-···)

   Fast and accurate React renderer for Notion. TS batteries included.

5. React Select (react-select.com/home)

   A flexible and beautiful Select Input control for ReactJS with multiselect, autocomplete, async and creatable support.

6. Sweet Alert (sweetalert.js.org)

   A beautiful replacement for success messages, error messages or info messages.

7. React Query (react-query.tanstack.com)

   Performant and powerful data synchronization for React.

8. React Hook Form (react-hook-form.com)

   Less code. More performant. Reducing the amount of code you need to write, and removing unnecessary re-renders are some of the primary goals of React Hook Form.

9. React Infinite Scroller (danbovey.uk/react-infinite…)

   Infinitely load content using a React Component.

10. React Motion (github.com/chenglou/react…)

    A spring that solves your animation problems. Level up your animation game.

11. React DnD (react-dnd.github.io/react-dnd/about)

    A React utility to help you build complex drag and drop interfaces while keeping your components decoupled.

12. React Desktop (reactdesktop.js.org)

    A JavaScript library built on top of Facebook's React library, which aims to bring a native desktop experience to the web, featuring many macOS Sierra and Windows 10 components.

13. Splitbee (splitbee.io)

    Track and optimize your online business with Splitbee. Your friendly analytics & conversion platform.

14. React Window (github.com/bvaughn/react-···)

    React components for efficiently rendering large lists and tabular data.

15. Stripe Elements (stripe.com/en-in/payments···)

    Stripe Elements are rich, pre-built UI components that help you create your own pixel-perfect checkout flows across desktop and mobile.

16. Chakra UI (chakra-ui.com)

    A simple, modular and accessible component library that gives you the building blocks you need to build your React applications.

17. Draft JS (draftjs.org)

    Rich Text Editor Framework for React.

    Draft.js fits seamlessly into React applications, abstracting away the details of rendering, selection, and input behavior with a familiar declarative API.

18. ChartJS (github.com/reactchartjs/r···)

    Build beautiful charts in minutes with Chartjs, Dashboards will looks sexy like never before.

19. React Slick (react-slick.neostack.com)

    React carousel component. One of the best sliders out there to showcase anything.

20. React Auto Suggest (react-autosuggest.js.org)

WAI-ARIA compliant autosuggest component built in React.

21. React Burger Menu (negomi.github.io/react-burger-m⋯)

    An off-canvas sidebar React component with a collection of effects and styles using CSS transitions and SVG path animations.

22. React Spinners (davidhu.io/react-spinners/)

    A collection of loading spinner components for React.

23. React Bootstrap Datatable (react-bootstrap-table.github.io/react-bootstra⋯)

    Next Generation of react-bootstrap-table. Datatable made easy like never before.

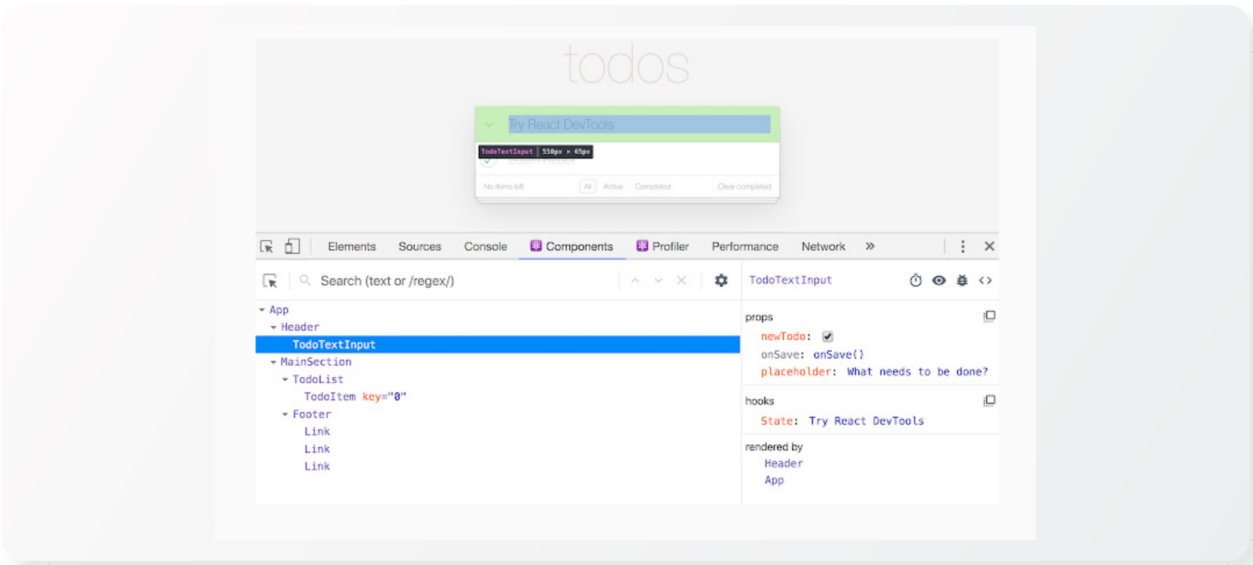
24. Styled Components (styled-components.com)

    styled components is the result of wondering how we could enhance CSS for styling React component systems.

25. Semantic UI (semantic-ui.com)

    Semantic is a development framework that helps create beautiful, responsive layouts using human-friendly HTML.

React Chrome Extension



```
▾ App
  ▾ Header
      TodoTextInput
  ▾ MainSection
    ▾ TodoList
        TodoItem key="0"
    ▾ Footer
        Link
        Link
        Link
```

```
TodoTextInput
props
  newTodo: ✓
  onSave: onSave()
  placeholder: What needs to be done?
hooks
  State: Try React DevTools
rendered by
  Header
  App
```

React Developer Tools is a Chrome DevTools extension for the React library. This extension allows us to inspect the React component hierarchies.

You can download it for free from this link.

You will get two new tabs in your Chrome DevTools

1. ⚛️ Components
2. ⚛️ Profiler

The Components tab shows the root React components rendered on the page. It will be a tree structure where you can inspect and edit the current props and state in the panel on the right.

What are some usecase of chrome extension?

1. Find out which component is rendering multiple times with the help of highlight updates.
2. Fake the data by changing the state/prop data if you want to visualize what UI will look like.
3. Analyze state/prop data at any point in time.
4. Find out which component is taking time to load in the profiler tab

Example to practices

Here are sample examples that you might love to work with

- Calculator Implementation of the iOS calculator built in React
- Emoji Search React app for searching emoji
- Snap Shot A photo gallery with search
- BMI Calculator A React Hooks app for calculating BMI
- Image Compressor An offline image compressor built with React and browser-image-compression
- Counter App A small shopping cart example
- Tutorial Solutions Solutions to challenges mentioned at the end of React tutorial
- Build Clone of Popular websites - Learn by watching other's code and then do it by yourself.

Want more examples?

Checkout amazing 100+ React Proof of Work Ideas from Fueler by Riten.

Interview Question

Cracking React interview is easy if you are good with fundamentals. Here are some of the resources you can refer to prepare yourself for the interview.

1. List of top 500 ReactJS Interview Questions & Answers
2. 500+ React Questions
3. Top 161 React interview questions and answers in 2021
4. React Interview Questions
5. Top 50 React Interview Questions You Must Prepare In 2022
6. 65 React Interview Questions

Wish you all the best for your interview, you are going to crush it.

Free Courses

Here are free courses that can be used to sharpen your ReactJS skills.

1. This course is for React newbies and anyone looking to build a solid foundation. It's designed to teach you everything you need to start building web applications in React right away.

   https://egghead.io/courses/the-beginner-s-guide-to-react

2. You'll develop a strong understanding of React's most essential concepts: JSX, class and function components, props, state, lifecycle methods, and hooks.

   https://www.codecademy.com/learn/react-101

3. This course is an excellent overview of "legacy" React class component development.

   https://egghead.io/courses/react-with-class-components-fundamentals-4351f8bb

4. Learn React's fundamentals with billionaires, fractal trees and live exercises.

   https://frontarm.com/courses/react-fundamentals/

5. The ultimate React 101 - the perfect starting point for any React beginner. Learn the basics of modern React by solving 140+ interactive coding challenges and building eight fun projects.

   https://scrimba.com/learn/learnreact

6. Udemy Free Courses

Note that these course keep on changing based on the active deals.

https://www.udemy.com/topic/react/free/

7. React Crash Course for Beginners 2021 - Learn ReactJS from Scratch in this 100% Free Tutorial!

   https://www.youtube.com/watch?v=Dorf8i6lCuk

8. React JS Crash Course

Get started with React in this crash course. We will be building a task tracker app and look at components, props, state, hooks, working with an API and more.

https://www.youtube.com/watch?v=w7ejDZ8SWv8

Thank you!

You have reached the end 🎉

If you liked this ebook content, Drop a rating at https://harshmakadia.gumroad.com/l/react-guide .

Follow Harsh Makadia and Dipen Dedania on Twitter for more such content.

> Cheers and Good luck to you! 🍀