

Overall Workings

<https://www.zipy.ai/blog/how-react-works-under-the-hood>

Reconciliation Process in Detail

<https://medium.com/@ruchivora16/react-how-react-works-under-the-hood-9b621ee69fb5>

React Under the Hood

React is a free and open-source front-end JavaScript library that aims to make building user interfaces based on components more seamless, [[Wikipedia](#)]. React is getting more and more popular nowadays and almost all companies that plan on building a groundbreaking front-end system have surely given a thought if they should be using React for their next groundbreaking app.

When I started building React Applications myself, I had never really paid attention to how React works under the hood, and I never really understood what the most common terms actually meant, i.e. Virtual Dom, LifeCycle of Components, DOM Updates etc. These weren't a concern for me at all and I just wanted to build my app fast. I started my journey working with functional components in React, and all I could care about were the things that could help me build faster i.e. hooks; for starters, all I cared about was how to use "props", "useState" and "useEffect". After all, these were the only things I needed to build my application and ship them to production.

But if you have ever deployed a React application, you might have already pointed out several issues that will arise due to the lack of fundamental knowledge. It would be just like building a car, just because you know the tire has to move and it needs gas and then you immediately jump to start building the car, and now viola! A whole new set of problem set has shown itself. Now you need a fuel pump because your initial assumption of just pouring the fuel into the engine and hoping it would work has backfired and your car is on a brink of explosion. Someway somehow you made the car run for a few kilometers just by chance, which is obviously very low and now the car can't make a turn because you don't know anything about a differential.

You see the point I am trying to make. Fundamental knowledge for building any system is necessary. So now let's get started straightaway.

JSX- JavaScript XML (Building Blocks of React)

Why do we use JSX when we already have HTML? Is JSX a subset of HTML for React? Does React require us to use JSX?

Do you know the answers to all these questions?

For reference, this is what the react docs state,

*JSX is a syntax extension for JavaScript that lets you write **HTML-like** markup inside a JavaScript file. Although there are other ways to write components, most React developers prefer the conciseness of JSX, and most codebases use it. [[React](#)]*

So now you might know the answer. Or maybe you have more questions now like what are the other ways? Let us dive deep into those questions. But before everything we need to know about

React Elements

A React Element is a small piece of code representing a part of the User Interface in a React Application.

Every React element is a JavaScript Object at the end.

Let us consider the following example:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>React Elements</title>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <div id="root"></div>
    <!-- React CDN -->
    <script
      crossorigin
      src="https://unpkg.com/react@18/umd/react.development.js">
  </script>
</body>
</html>
```

```

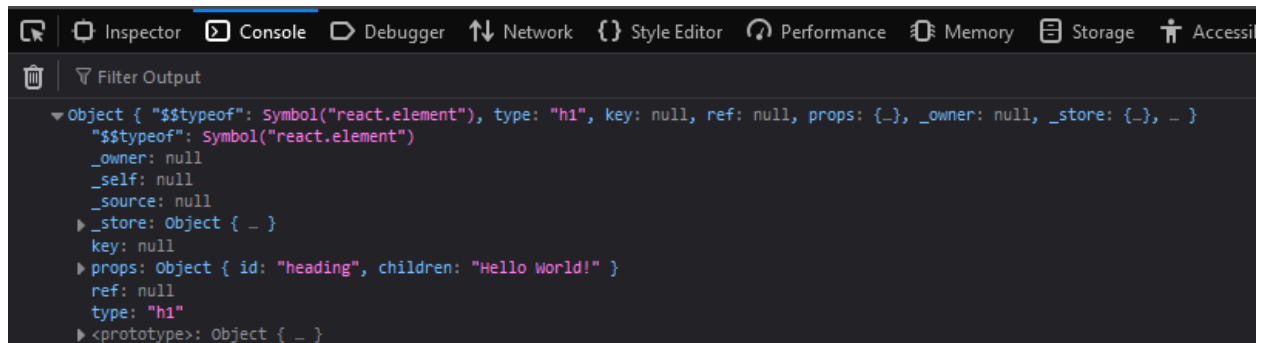
</script>
<script
  crossorigin
  src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
</script>

<!-- React Code -->
<script>
  const heading = React.createElement(
    "h1",
    { id: "heading" },
    "Hello World!"
  );

  const root = ReactDOM.createRoot(document.getElementById("root"));
  root.render(heading);
</script>
</body>
</html>

```

The above example shows a basic react application that creates a “h1” tag saying “Hello World!”. Now what does this create element look like in the console? Let’s see that too.



As you can see this returns a type of “Object”. So let’s recall once again.

Every React element is a JavaScript Object at the end.

A ReactElement is simply a representation of a DOM element in the Virtual DOM.

Answer this.

Do browsers natively support JS objects and render them as HTML?

No my dear friends, the Browser does not support JS Objects as HTML elements. We use the ***root.render()*** method from ReactDOM to convert the JS Objects into HTML elements, which can then be rendered onto the browser.

NOTE: *root.render(document.getElementById("root")) replaces the element inside the id "root".*

Now you know the fundamental concepts of React and ReactDOM and why they are used. So now you're ready to build a real-world application using React Elements. But wait, in a web app, a single element is never displayed on our webpage so let's try to build a nested structure using "React.createElement". Our HTML will look something like this

```
/**
 *
 * <div id="parent">
 *   <div id="child">
 *     <h1> I am h1 tag </h1>
 *     <h2> I am h2 tag </h2>
 *   </div>
 *   <div id="child2">
 *     <h1> I am h1 tag </h1>
 *     <h2> I am h2 tag </h2>
 *   </div>
 * </div>
 */
```

Now let us write the code in Legacy React

```
const parent = React.createElement("div", { id: "parent" }, [
  React.createElement("div", { id: "child" }, [
    React.createElement("h1", {}, "I am an h1 tag"),
    React.createElement("h2", {}, "I am an h2 tag"),
  ]),
  React.createElement("div", { id: "child2" }, [
    React.createElement("h1", {}, "I am an h1 tag"),
    React.createElement("h2", {}, "I am an h2 tag"),
  ]),
]);
```

You can see right, how tedious everything is starting to get. Just for a simple HTML as shown above we have to write so much code. Suppose you manage to deploy the application since you're a 10x developer and of course, you can manage that but now a junior developer who just joined your team has to work on this code on a production website. You can imagine what would happen next.

Here comes JSX

```
const jsxHeading = <h1 id="heading"> This is a H1 tag</h1>
```

First of all, remove the pre-conception that JSX is HTML inside JavaScript.

JSX aims to combine HTML and JavaScript. JSX is **HTML-like** but not HTML inside JS.

JSX is a React Element i.e. a JS Object

When you write something like:

```
const greet = <h1> Hello </h1>
```

what you're essentially doing is this:

```
const greet = React.createElement("h1", {}, "Hello");
```

If you console and see the difference between **heading** and **jsxHeading**, there will be no difference. And since JSX is very easy to understand and remember unlike `React.createElement`, we use JSX throughout React application.

So let's repeat once again. *JSX is not HTML inside JS but is HTML-like syntax. JSX is a JavaScript Object i.e. `React.createElement()` at the end of the day.*

And now the curious folks might ask, since JSX is an HTML-like syntax element how is it a JavaScript object? And how is it rendered on my Browser if it's not HTML?

Well, have you heard of Babel? Yes, that's right; It's Babel's job to transpile (convert) the HTML-like JSX code into *React Elements* before it reaches the JS engine.

You can see this conversion yourself on [Babel's official website](#)

Put in next-gen JavaScript	Get browser-compatible JavaScript out
<pre><h1>Hello</h1></pre>	<pre>/*#__PURE__*/React.createElement("h1", null, "Hello");</pre>

Why do we need to convert it into *React Elements*?

Because JS engines do not natively support JSX and can only work with ECMAScript syntaxes and nothing else; so we need Babel to transpile the JSX into React Elements which can be read by the JS engine (afterall React Elements are JS Objects). And now the **render method of ReactDOM** can work its magic and convert the JS Objects / **React Elements** into HTML that can be then understood and then be rendered by our browsers.

We also need to use JavaScript inside JSX, so to do that we simply wrap it around “{ }”

```
<p> Sum {1 + 1} </p>
```

This results in

Sum 2

That's it. So now you've got a pretty good idea about why we use JSX and why is it used everywhere in React applications.

React Components

React components are the core building blocks of a React application that represent how a particular element would be visualized on the User Interface. There are two types of Components in React.

1. Functional Components
2. Class Based Components

Functional components are what Modern React Applications use. React suggests we use functional components after React 16.8. Class-based components, however, is the legacy way of declaring components which help us track the lifecycle of the components more accurately.

Let's see examples of how Functional and Class Based Components are declared in React.

Functional Components

Functional Components are nothing but normal JavaScript functions that return a piece of JSX code or a React Element.

```
const App = () => {  
  return (  
    <div className="app">  
      <h1>App Component</h1>  
    </div>  
  );  
};
```

As you can see, our functional component is a normal JS function that returns a piece of JSX. That's it.

Class-Based Components

Class-based components is a JavaScript Class that has a *render method* which returns a piece of JSX or React Element.

```
class About extends React.Component {  
  render() {  
    return (  
      <div className="app">  
        <h1>App Component</h1>  
      </div>  
    );  
  }  
}
```

So once again, class-based components have a render method that returns a piece of JSX instead of returning the JSX directly like in Functional Components.

*The difference between class-based and functional components is how they return JSX or React Element. Class-based components have a **render method** that returns a piece of JSX whereas Functional Components return the piece of JSX.*

Declaring state and using props is also different in both Components but we will discuss more about that later.

Rendering a Component on the screen

To render a component on the screen, we use the **render()** method provided to us by ReactDOM. We directly add our component inside the render method and since we have to use JSX to render a component, we represent the component as `<Component />`.

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";

const rootElement = document.getElementById("root")!;
const root = ReactDOM.createRoot(rootElement);

root.render(<App />);
```

Component Composition

A jargon used in the industry

It refers to using two or more components with each other. Our App Component uses Component Composition when it consists of other components within it.

```
export default function App() {
  return (
    <div className="App">
      <User name="John Doe" email="xyz@xyz.xyz" />
    </div>
  );
}
```



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

In the above example, the App component has a User component. And when two or more components are used to create a component, it is known as component composition.

Props

Props are short for properties, and they are used to pass data between components. Passing props is an essential process of React Applications as wherever we need to share data between multiple components we use props. So let us now see how we can do it in Functional and Class based components.

Let us pass User details from our App Component to our User Component.

Passing props to Functional Component

```
function App() {
  return (
    <div className="App">
      <User name="John Doe" email="xyz@xyz.xyz" />

    </div>
  );
}
```

Here we pass name and email as props. We pass them as arguments to our component. Now let us try to access these props within our **User** component.

```
const User = (props) => {
  return (
    <div>
      <p> Full Name: {props.name} </p>
      <p> Email: {props.email} </p>
    </div>
  );
};
```

```
};  
  
export default User;
```

We can access the props directly in our Functional Component's Parameters as an object. And get the value of the **props** object directly from its key.

Remember the key is the argument passed from the **App** Component, i.e. the **Parent component**.

Or we can directly destructure the object and get the keys, this approach is used widely in the industry.

```
const User = ({ name, email }: { name: string; email: string }) => {  
  return (  
    <div>  
      <p> Full Name: {name} </p>  
      <p> Email: {email} </p>  
    </div>  
  );  
};  
  
export default User;
```

Passing props to Class-Based Component

Passing the prop to our child components is the same, both in Functional and Class-based components. The difference is how we access these props. Let us see in detail.

```
function App() {  
  return (  
    <div className="App">  
      <User name="John Doe" email="xyz@xyz.xyz" />  
    </div>  
  );  
}
```

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

In class-based components, we can access the props by using the constructor to initialize the props with the instance of our Class.

```
import React from "react";

class User extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <p> Full Name: {this.props.name} </p>
        <p> Email: {this.props.email} </p>
      </div>
    );
  }
}

export default User;
```

An Important Question

Why do we use **super(props)** inside our constructor?

- We use `super(props)` to ***initialize the parent component***, i.e. call the constructor of the parent class so that no unwanted behavior occurs in our app. Since we have used inheritance and extended our class-based component we need to make sure the Parent Component i.e. **React.Component** is properly initialized.
- The unwanted behavior typically includes:
 1. ***“this.props” being undefined in the constructor***: This can lead to errors if you try to access or manipulate props in the constructor itself.
 2. ***Inconsistent behavior***: While “this.props” might still be accessible in other lifecycle methods (like `render`), it will not be initialized in the constructor, leading to confusion and potential bugs.

React even provides a warning in the console if you forget to pass props to `super()` in the constructor:

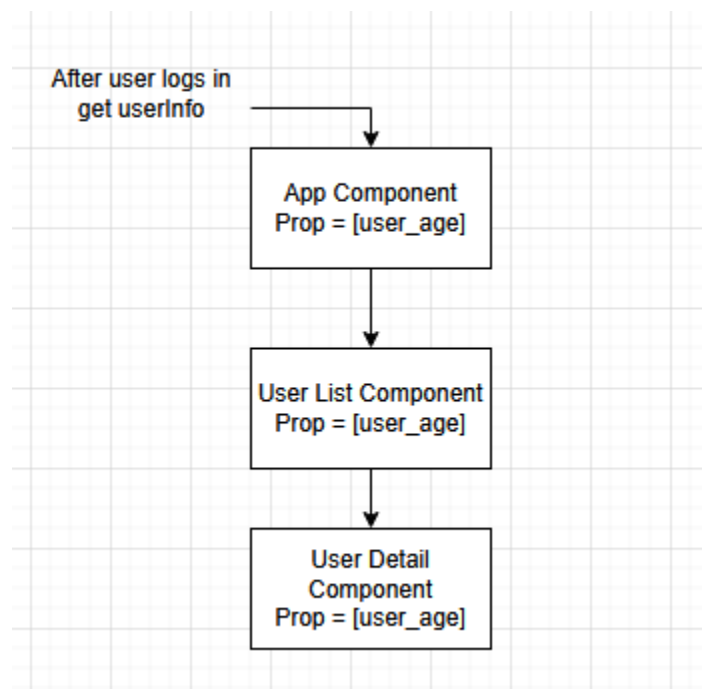
"Warning: When calling `super()` in a component, make sure to pass up the same props that your component's constructor was passed"

Using **"`super(props)`"** ensures that **"`this.props`"** is properly initialized and avoids runtime errors or warnings.

Prop Drilling

As you have already seen, we can pass props into child components by passing them as arguments of the Component and then access it from the parameter. But suppose we need the prop **n-levels** deep.

Let us consider a basic example where a user logs in and after that, we need to get the user's details. Let us only consider "age" for now, and we need to pass this age to the UserDetail Component. Then the flow would be



As you can see if we need the user's details we need to pass the details as props from our top-level App Component to the User Detail Component. Now imagine if this chain is longer and we need to pass every detail of the User. This approach may work if you have

to just read the data, but if you need to modify the data, then you can imagine the inefficiency of doing so.

This drilling of prop through layers of components is known as Prop Drilling. So how do we prevent this Prop Drilling and manage our application in an efficient and maintainable way?

The Answer

Using a *State Management System like Redux or Context*. But wait before that, what exactly is a state?

State Variables

State variables are the “memory” of Components in our React application. These variables allow React components to manage and store data that changes over time.

Typing into the form should update the input field, clicking “next” on an image carousel should change which image is displayed, and clicking “buy” should put a product in the shopping cart. In these examples, the components need to “remember” things: the current input value, the current image, and the shopping cart. In React, this kind of component-specific memory is called a state. [State in React](#)

So as you already have seen, state variables are a crucial part of React, which provides interactivity to our application. So how do we define state variables?

Using State Variables in Functional Components

In Functional Components, we use **HOOKS** to hook into the superpowers that react gives us. In this case, we use the “**useState**” hook to declare the state variable. Let’s see it in action

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

const User = ({ name, email }: { name: string; email: string }) => {
  const [count, setCount] = useState(1);
  return (
    <>
      <div>
```

```

    <p> Full Name: {name} </p>
    <p> Email: {email} </p>
  </div>
  <div>
    <h3>Current Level:{count}</h3>
    <button onClick={() => setCount(count + 1)}>Increase
level</button>
  </div>
</>
);
};

export default User;

```

We use the `useState()` hook to define a state. The `useState()` returns a state variable and a `setState` dispatch action to update the state.

Inside the `useState` method, we provide an initial value which can be null, integer, string, object, array, or any other datatype. In this case, we set the initial value of the count to 1.

To access the state we use the state name directly, in this case, `count`.

Modifying the state requires us to use the `setCount` method where we need to update the state value. As shown in the above example we use “`setCount`” to modify the state.

```

<button onClick={() => setCount(count + 1)}>Increase level</button>

```

Note: We cannot update the state directly by using `count + 1`.

Why?

Because doing so would bypass React's state management system and could lead to unpredictable behavior and rendering issues; essentially, we should treat the state as **immutable** and always create a new copy when updating it. And the “`setCount`” dispatch action (`setCount`) does just that.

Key points about updating the state:

- **Immutability:** React expects the state to be treated as immutable data, meaning you should not directly change the values within a state object.
- **setState function:** To update the state, you must use the setState function (**setCount** in this case) which allows React to track changes and re-render the component accordingly.

Potential problems with direct modification:

- No re-render: If you directly modify a state variable, React might not detect the change and won't re-render the component, leading to visual inconsistencies.

Question

We declare state variables like this.

```
const [count, setCount] = useState(1);
```

But in JavaScript, we cannot modify a constant so how is React assigning a new value to a constant?

Answer

When we update our state, the entire component is being re-rendered / re-initialized. Thus breaking the link between the previous value and the new value. This means that after the component is re-rendered/ re-initialized, the previous state variable which we had created using **const** is now no longer available and we now have a new instance of the state constant initialized with a new value.

Creating a new instance of the component means that there is no link between the old constant and the new constant thus enabling us to assign a new value to a constant.

Using State Variables in Class-Based Components

In class-based components, we define the state variables as an object inside the constructor. This object can be however big we want and React knows the given object is a state when we define it inside the “**this.state**” object. Let's see it in action.

```
import React from "react";

class User extends React.Component {
```

```

constructor(props) {
  super(props);
  this.state = {
    count: 1,
  };
}

render() {
  return (
    <>
      <div>
        <p> Full Name: {this.props.name} </p>
        <p> Email: {this.props.email} </p>
      </div>
      <div>
        <h3>Current Level:{this.state.count}</h3>
        <button
          onClick={() => this.setState({ count: this.state.count + 1 })}
        >
          Increase level
        </button>
      </div>
    </>
  );
}
}

export default User;

```

To access the count state we use “**this.state.count**”, this.state is the state object and count is the state variable that we recently defined. Same as getting the value of an object from its key.

To modify the state in class-based components, it is the same as in functional components where we do not modify the state directly but we use the **setState** method to modify the state variable.

```

<button onClick={() => this.setState({ count: this.state.count + 1 })} >
  Increase Level

```



```
</button>
```

Here “**this.setState**” method takes an object, where the key is the state variable and then we modify the state here itself. To get the current state value, just like in the above scenarios we use “**this.state**” and update the state variable accordingly.

Now do you know why we cannot modify the state variables directly?

Responding to Events

While updating a state, the proper way of calling the “**setState**” method is given as shown below.

Let us declare a state as

```
const [count, setCount] = useState(1);
```

And to update the state we do,

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

const User = ({ name, email }: { name: string; email: string }) => {
  const [count, setCount] = useState(1);

  const handleIncrementCounter = () => {
    setCount(count + 1)
  }
  return (
    <>
      <div>
        <p> Full Name: {name} </p>
        <p> Email: {email} </p>
      </div>
      <div>
        <h3>Current Level:{count}</h3>
        <button onClick={handleIncrementCounter}>Increase level</button>
      </div>
    </>
  );
};
```

```
export default User;
```

Can you tell the difference between

```
<button onClick={handleIncrementCounter}>Increase level</button>
```

```
<button onClick={() => handleIncrementCounter(incrementStep)}>Increase  
level</button>
```

```
<button onClick={handleIncrementCounter(incrementStep)}>Increase  
level</button>
```

The Answer

```
<button onClick={handleIncrementCounter}>Increase level</button>
```

- This directly assigns the function **handleIncrementCounter** to the `onClick` event handler. When the button is clicked the function **handleIncrementCounter** will be called without any arguments.

```
<button onClick={() => handleIncrementCounter(incrementStep)}>Increase  
level</button>
```

- This approach uses the arrow function to create an anonymous function that calls the **handleIncrementCounter** with the “**incrementStep**” argument when the button is clicked.
- This is the correct way to pass arguments to Event Handlers.

```
<button onClick={handleIncrementCounter(incrementStep)}>Increase  
level</button>
```

- This immediately invokes the **handleIncrementCounter** function with the “**incrementStep**” argument when the component renders. *Not when the button is clicked but during component render.*

- INCORRECT WAY to use Event Handler as it will call the function right away, leading to unwanted execution of the function.

Life Cycle of React Components

In React, the lifecycle of a component refers to the various stages a component goes through while displaying the user interfaces.

Let's understand what happens when a component is rendered on a webpage.

1. Mounting: Component needs to be loaded on the screen
2. Updating: When state changes, those changes need to be updated on the screen.
3. Unmounting: Finally when we navigate away from the component, the component needs to be unmounted from the screen.

These lifecycle stages allow developers to control and run specific code at key moments, i.e. when a component is created (Mounted), updated, and removed (Unmounted).

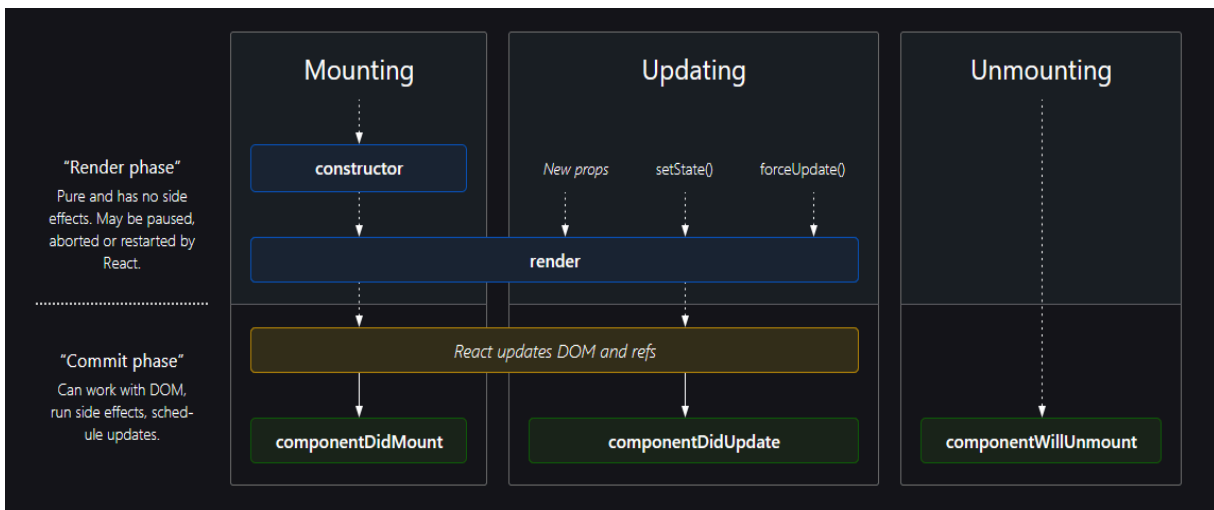


Fig: Lifecycle of React Components

[View the diagram here](#)

Upon closer inspection of the above diagram and analyzing the different phases. You might now have a general idea of why React is fast. There are two phases during a component's lifecycle.

1. **Render Phase:** *Pure and has no side effects. May be paused, aborted, or restarted by React.*
2. **Commit Phase:** *Can work with DOM, run side effects, and schedule updates.*

So what does this mean?

Render Phase

During the render phase, pure functions i.e. functions that return the same output for the same sets of input, in this case, the JSX for displaying the React Component is rendered first.

But my JSX is dependent on my state value, which I get from the API.

Yes, this is the second stage and we will discuss this in the commit phase. But before the data is fetched from the API, there will be a loading or a shimmer screen, right? Even if there is no loading screen or a Shimmer UI, on some websites you might have seen that while the page loads, first the layout appears, and then after some time, the data populates.

This is exactly what the **Render Phase** does. It allows React to load pure components first providing a seamless user experience without worrying about the data layer.

Commit Phase

After the *Render Phase* comes the *Commit Phase*, i.e. working with DOM, running side effects, and scheduling updates. For instance, whenever we request an API, we need to render the result.

The commit phase does just that. After the render phase is over and the component is loaded on the screen, we need to update it with the appropriate values. The Commit phase allows us to update these values onto the DOM. Other than updating the values onto the DOM, we can also set up a subscription, and manually change the DOM in React components.

Loading the UI onto the screen

Generally, there are two approaches to loading the UI onto the screen while making an API request

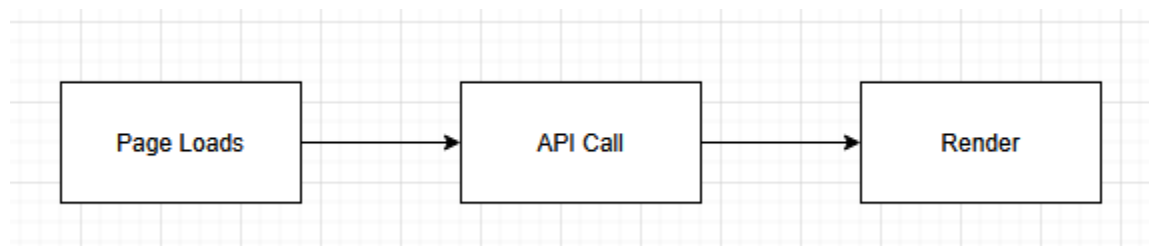


Fig: Option A

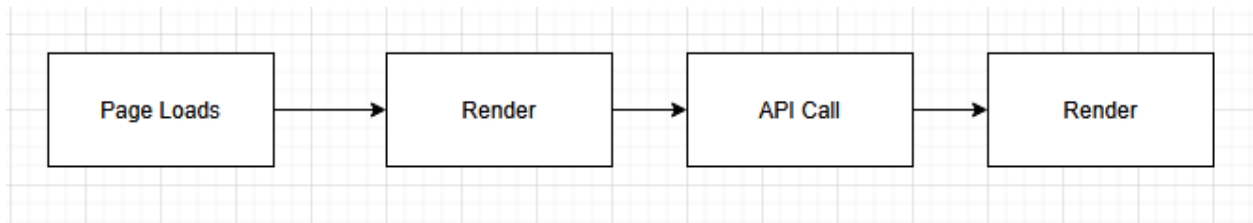


Fig: Option B

Which approach do you think provides a better UX? Of the above two approaches to loading the UI while making an API request, which approach do you think React uses? (Compare them with the lifecycle of the React Components diagram)

Yes, It's the 2nd approach.

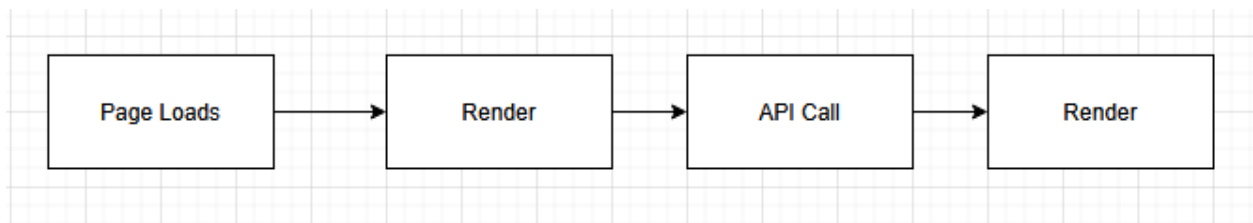


Fig: Option B

There are two render phases in the above diagram, now after knowing about the lifecycle phases and steps to load the UI onto the screen, can you tell where the **Render** and **Commit** phase fit into the above diagram?

In this article, we will only be discussing about 3 Lifecycle Methods. If you understand these three methods then you can breeze through the other lifecycle methods as well. We will be using class-based components as examples to help us understand the various lifecycle stages and it's internal workings. So let's get started.

ComponentDidMount

This method is called after the component is rendered in the DOM. It is the perfect place to make API calls, set up subscriptions, add event listeners, and do anything that needs to run side effects and schedule updates.

```
componentDidMount(): void {  
  // Make API CALL HERE  
  console.log("Component mounted");  
}
```

```
}
```

The “**componentDidMount**” method is **only called once**, during the initial render. After the component has been mounted onto the screen, any changes made to the data layer are handled by “**componentDidUpdate**”.

ComponentDidUpdate

This method is used to keep track of state changes.

```
componentDidUpdate(prevProps: Readonly<{}>, prevState: Readonly<{}>): void {
  {
    if (this.state.count !== prevState.count) {
      console.log("State Changed")
    }
  }
}
```

As you can see we can compare the state changes in the **ComponentDidUpdate** method and perform the updates here.

ComponentWillUnmount

This method is called when we leave the page or when the element is removed from the DOM.

```
componentWillUnmount(): void {
  // Run clean up here
  console.log("Component unmounted")
}
```

We generally use **componentWillUnmount** to run the clean-up function. More will be discussed later.

Now let us dive in deeper. To better understand the lifecycle of React Component, as stated earlier, we will be using Class-Based Components as an example.

Examples

Let us consider a Parent and a Child Component. We need to render both the Parent and the Child Component which consists of a Counter (Inside the Parent Component) and a button to update the counter value (Inside the Child Component). We are trying to update the counter (state) of the Parent Component using the Child Component, i.e. Lifting the State Up.

We will gradually move toward the above-stated problem but first, let us just see what lifecycle methods run when only the bare components are rendered.

Parent Component

```
import React from "react";

import ChildComponent from "../ChildComponent";

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

  componentDidMount(): void {
    console.log("Parent Component mounted");
  }

  componentDidUpdate(): void {
    console.log("Parent Component updated")
  }

  componentWillUnmount(): void {
    console.log("Parent Component unmounted")
  }

  render(): React.ReactNode {
    console.log("Parent Component render")
    return (
      <>
        <div>Parent Component</div>
        <br/>
      </>
    )
  }
}
```

```

        {/* Child Components Rendered Here*/}
        <ChildComponent />
    </>
    );
}
}

export default Parent;

```

Child Component

```

import React from "react";

class Child extends React.Component {
  constructor(props) {
    super(props);
    console.log("Child Component constructor")
  }

  componentDidMount(): void {
    console.log("Child Component mounted");
  }

  componentDidUpdate(): void {
    console.log("Child Component updated")
  }

  componentWillUnmount(): void {
    console.log("Child Component unmounted")
  }

  render(): React.ReactNode {
    console.log("Child Component render")
    return (
      <div> Child Component </div>
    );
  }
}

```



```
export default Child;
```

During the initial render, what do you think the output for the above code will be in the console?



The screenshot shows a web browser's developer console with the following log entries:

- Console was cleared
- Parent Component constructor
- Parent Component render
- Child Component constructor
- Child Component render
- Child Component mounted
- Parent Component mounted

If you correctly remember about the **Render** and the **Commit** phase discussed earlier, the process is pretty straightforward, right? Let's revise.

First, the Render phase is initiated and only then comes the commit phase, i.e. the components need to be loaded on the screen first and only then the data is fetched.

So what would the overall process look like?

1. First, the parent component is displayed on the screen. So its constructor is called and then its render method.
2. While rendering the Parent Component, React encounters the child component, so now it starts to render the child component, so now the constructor and the render method of the Child Component are initialized.
3. After the Child Component's **Render phase** is over, it goes on to the **Commit phase**, so it calls the "**componentDidMount**" method and logs "Child Component mounted" onto the screen.
4. Now that the child component is fully rendered, the parent's component **Render phase** is over. So all that's remaining is the parent's component **Commit phase**, so now it calls its "**componentDidMount**" method and logs "Parent Component mounted" onto the screen.

Pretty straightforward right? After understanding the different phases it's quite easy, right? So now let's look at a few more examples to test our knowledge and guess the output.

Example 1

Let us create a **“userName”** state inside the Parent Component. The **“userName”** state is updated from the Parent Component itself and is then rendered into the Child Component.

Parent Component

```
import React from "react";
import ChildComponent from "../ChildComponent";

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

    console.log("Parent Component constructor");
    this.state = {
      userName: "",
    };
  }

  componentDidMount(): void {
    console.log("Parent Component mounted");
    this.setState({
      userName: "The StackSmith",
    });
  }

  componentDidUpdate(): void {
    console.log("Parent Component updated");
  }

  componentWillUnmount(): void {
    console.log("Parent Component unmounted");
  }
}
```

```

render(): React.ReactNode {
  console.log("Parent Component render");
  return (
    <>
      <div>Parent Component</div>
      { /* Child Components Rendered Here */ }
      <ChildComponent
        userName={this.state.userName}
        count={this.state.count}
        onCounterUpdate={this.handleChildCounterUpdate}
      />
    </>
  );
}
}

export default Parent;

```

Child Component

```

import React from "react";

class Child extends React.Component {
  constructor(props) {
    super(props);
    console.log(this.props.userName, "Child Component constructor");
  }

  componentDidMount(): void {
    console.log(this.props.userName, "Child Component mounted");
  }

  componentDidUpdate(): void {
    console.log(this.props.userName, "Child Component updated");
  }

  componentWillUnmount(): void {

```

```

    console.log(this.props.userName, "Child Component unmounted");
  }

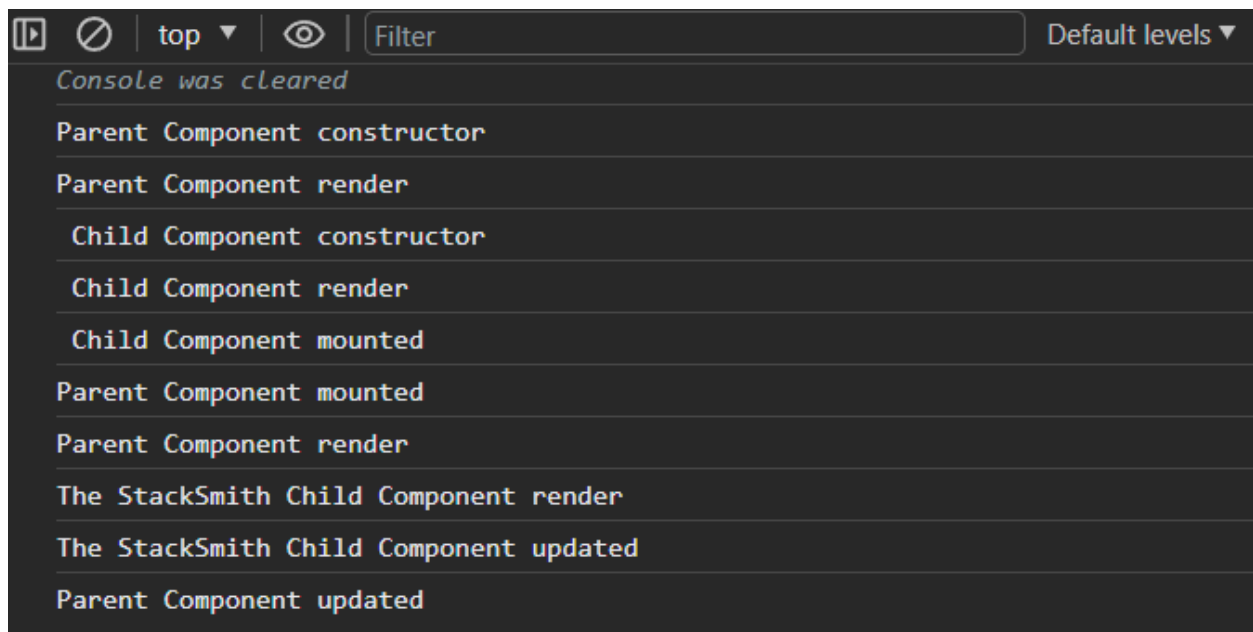
  render(): React.ReactNode {
    console.log(this.props.userName, "Child Component render");
    return (
      <div>
        Child Component
      </div>
    );
  }
}

export default Child;

```

Think of the output of the above code for a while.

The Answer



Let's see what happens now

1. Firstly during the **render phase**, the parent component is rendered onto the screen, i.e. Parent's constructor and render method are called.
2. Then the child component needs to be rendered, so its constructor and render method are called.

Since the props haven't been initialized i.e. "this.props.userName" does not have a value yet, there is a whitespace at the beginning of the console.log().

3. Now the **render phase** is over and we move onto the **commit phase**.
4. So now the **Child Component's componentDidMount** method is called and then the **Parent Component's componentDidMount** is called.
5. When the **Parent Component's componentDidMount** is called, the state changes and React needs to display the latest changes.

NOTE: After an update operation, React again re-renders the component.

6. So now the Parent Component's render method is called again subsequently calling the Child Component's render method as well and logs the following.
 - a. Parent Component render
 - b. The StackSmith Child Component render

Since the component has been updated with a new value, the child component now calls its **componentDidUpdate** method, instead of the "componentDidMount" method.

- c. The StackSmith Child Component updated
7. Similar to how the Child Component called its "**ComponentDidUpdate**" method after the prop value was updated, the Parent Component also calls its "**ComponentDidUpdate**" method after the child component has finished rendering.

In this second example, we got into more detail about the lifecycle methods, right? Specially about how the **ComponentDidUpdate** method works and how it tracks the changes throughout our application. Now let's go into more detail

Example 2

Let us add another "**count**" state inside the Parent Component. The "**count**" state is incremented from the child component using "**handleCounterUpdate**". [*Lifting the State Up*]

Parent Component

```
import React from "react";
```

```
import ChildComponent from "../ChildComponent";

class Parent extends React.Component {
  constructor(props) {
    super(props);
    console.log("Parent Component constructor");

    this.state = {
      userName: "",
      count: 0,
    };

    this.handleChildCounterUpdate =
this.handleChildCounterUpdate.bind(this);
  }

  componentDidMount(): void {
    console.log("Parent Component mounted");
    this.setState({
      userName: "The StackSmith",
    });
  }

  componentDidUpdate(): void {
    console.log("Parent Component updated");
  }

  componentWillUnmount(): void {
    console.log("Parent Component unmounted");
  }

  handleChildCounterUpdate() {
    this.setState((prevState) => ({
      count: prevState.count + 1,
    }));
  }

  render(): React.ReactNode {
    console.log("Parent Component render");
    return (
```

```

    </>

    <div>Parent Component</div>
    { /* Child Components Rendered Here */ }
    <ChildComponent
      userName={this.state.userName}
      count={this.state.count}
      onCounterUpdate={this.handleChildCounterUpdate}
    />
  </>
);
}
}

export default Parent;

```

Child Component

```

import React from "react";

class Child extends React.Component {
  constructor(props) {
    super(props);
    console.log(this.props.userName, "Child Component constructor");
  }

  componentDidMount(): void {
    console.log(this.props.userName, "Child Component mounted");
  }

  componentDidUpdate(): void {
    console.log(this.props.userName, "Child Component updated");
  }

  componentWillUnmount(): void {
    console.log(this.props.userName, "Child Component unmounted");
  }
}

```

```

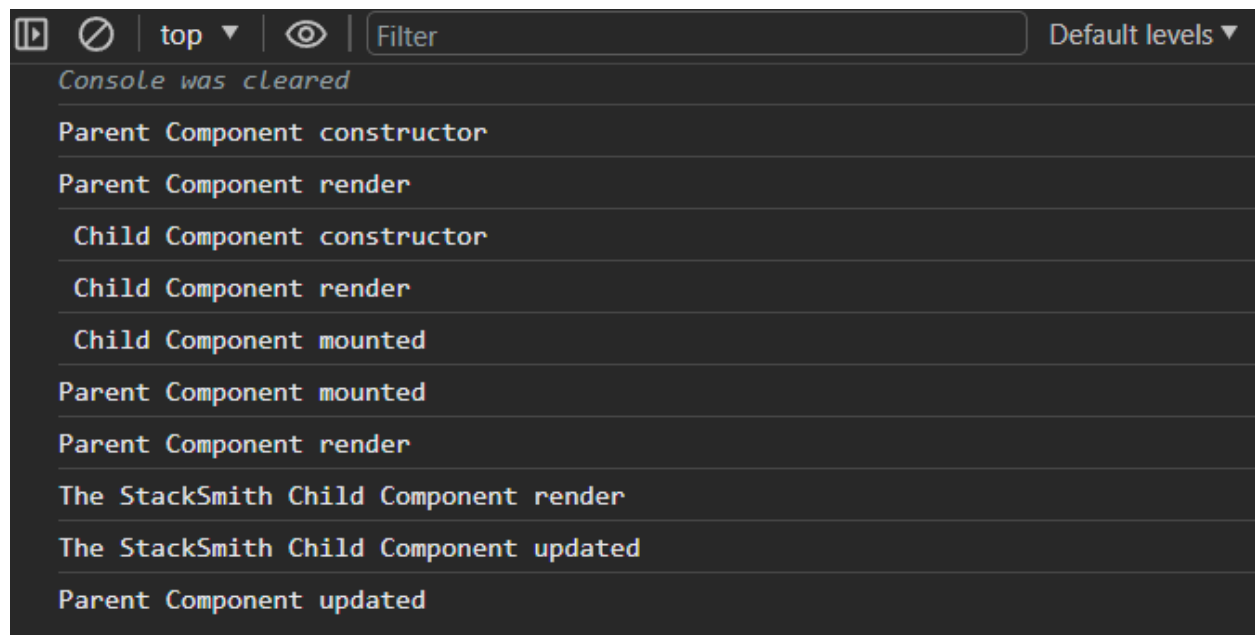
    handleCounterUpdate = () => {
      this.props.onCounterUpdate(); // Call the passed function
    };

    render(): React.ReactNode {
      console.log(this.props.userName, "Child Component render");
      return (
        <div>
          <br />
          Count: {this.props.count} <br />
          <button onClick={this.handleCounterUpdate}>Increase
counter</button>
        </div>
      );
    }
  }
}

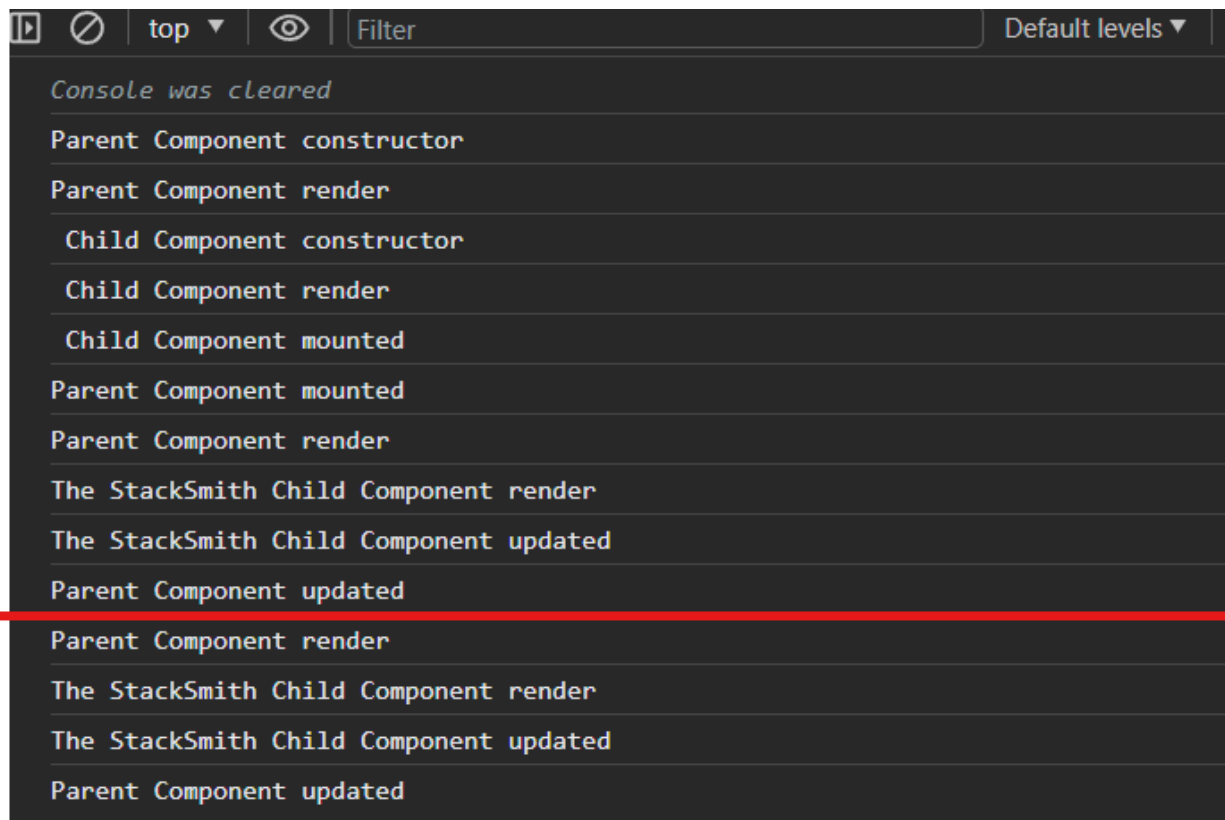
export default Child;

```

The initial output after the first render is the same as before



But what will happen when we click the **Increase Counter** button?



```
Console was cleared
Parent Component constructor
Parent Component render
  Child Component constructor
  Child Component render
  Child Component mounted
Parent Component mounted
Parent Component render
  The StackSmith Child Component render
  The StackSmith Child Component updated
Parent Component updated
Parent Component render
  The StackSmith Child Component render
  The StackSmith Child Component updated
Parent Component updated
```

Yes, it's the exact operations as before.

Summary

1. After the **Parent Component's componentDidMount** is called, the state changes and React needs to display the latest changes.

After an update operation, React again re-renders the component.

2. So now the Parent Component's render method is called again subsequently calling the Child Component's render method as well, and logs the following.
 - a. Parent Component render
 - b. The StackSmith Child Component render

Since the component has been updated with a new value, the child component now calls its **ComponentDidUpdate** method.

- c. The StackSmith Child Component updated
3. Similar to how the Child Component called its "**ComponentDidUpdate**" method after the prop value was updated, the Parent Component also calls its "**ComponentDidUpdate**" method after the child component has finished rendering.

4. After the Increment Counter button is clicked, these steps are again repeated.

Tricky Interview Question

Suppose we have a `setInterval` in the **`componentDidMount`** method.

```
componentDidMount(): void {  
  this.timer = setInterval(() => {  
    console.log( "Child Component mounted");  
  }, 1000)  
}
```

Now after the component with the interval is loaded we move away from the current page. What do you think will happen? Will the timer reset?

No, the timer will not reset and continue to run in the background. If we move away from the current page, i.e. unmount the component and move back to the current page with the Interval Timer, the timer method will mount again, thus creating two timers that will log “Child Component mounted” on the screen at different intervals, i.e. 500ms. And if we move back and forth between the pages, the timer will keep on mounting on the page instead of resetting.

This causes a lot of memory leaks and performance issues.

Now you might already know which lifecycle method we will use here to free up the memory right?

```
componentWillUnmount(): void {  
  clearInterval(this.timer)  
}
```

The “**`componentWillUnmount`**” method is usually used to clear up the memory after the component is unmounted from the screen. This usually involves clearing the timer, the Interval, or any other tasks that may cause memory leakage if not handled properly.

Never Compare React Lifecycle Methods To Functional Components

Even I have been guilty of this. I had learned functional components at first and while studying Lifecycle Methods, I tried comparing them with functional methods, hoping that it would make my understanding better. But comparing different lifecycle methods led me to “**useEffect**” hook every time.

I am not saying we can't relate to it but understanding “**componentDidMount**”, “**componentDidUpdate**”, “**componentWillUnmount**”, and other lifecycle methods separately and then relating them is a better approach and will strengthen our foundation even better.

Now since we have already covered the lifecycle methods. Let's look at some of the examples now.

Suppose in the class-based method,

1. We need to run the “**componentDidMount**” method, In “**useEffect**” we do

```
useEffect ( (e)=>{  
  console.log( "Child Component mounted")  
}, [])
```

When we leave the dependency array blank, it only runs once while the page loads.

But if we do this,

```
useEffect ( (e)=>{  
  console.log( "Child Component mounted")  
})
```

i.e. omit the dependency array. The “**useEffect**” runs after every render, and you will not be completely wrong to think of this as a form of “**componentDidUpdate**”.

2. We need to keep track of the userInfo object and see if any changes have been made. How would we do that?

In class-based components, we would do

```

componentDidUpdate(prevProps: Readonly<{}>, prevState: Readonly<{}>):
void {
  if (this.state.userInfo.name !== prevState.userInfo.name) {
    console.log("State Changed")
  }
  if (this.state.userInfo.email !== prevState.userInfo.email) {
    console.log("State Changed")
  }
  if (this.state.userInfo.phone !== prevState.userInfo.phone) {
    console.log("State Changed")
  }
}

```

You can see how cumbersome this is. Now how would we keep track of the `userInfo` object using the “`useEffect`” hook?

```

useEffect((e)=>{
  userInfo.name = e.target.value
}, [userInfo])

```

And if we needed to track more objects we would just write another “`useEffect`” for it.

3. Now how would we handle the “`componentWillUnmount`”, to free up our memory

```

useEffect(()=>{
  const timer = setInterval(()=>{
    console.log("Timer")
  }, 1000);

  return clearInterval(timer)
}, [])

```

Using the `return` keyword, we will handle the “`componentWillUnmount`” method.

As you can see, everything is handled by “`useEffect`”, but while learning at first and trying to understand the lifecycle we have to think about whether to leave the

dependency array blank, not include it, or add a state variable to it, and even when to use the return keyword. And if we mess it up unknowingly we might just spin around in circles in parts where we simply didn't have to. So understanding lifecycle methods separately and then relating it to the functional component equivalence is the best approach we can follow while learning the lifecycle methods 😊.

Few Questions

1. *Do you know why we use `super(props)`?*
2. *Why are component `mount`, `update`, and `unmount` being called at last after the page loads?*
3. *When a component updates, does the state DOM update to reflect the changes or does the entire component re-render?*

How does React work?

Before getting into the nitty-gritty details, we must first know about

1. Components
2. State
3. Props

Understanding Document Object Model (DOM)

DOM stands for Document Object Model, which is a programming interface for web documents. It treats an HTML or XML document as a tree structure wherein each node is an object representing a part of the document. It is used to connect web pages to programming languages and scripts.

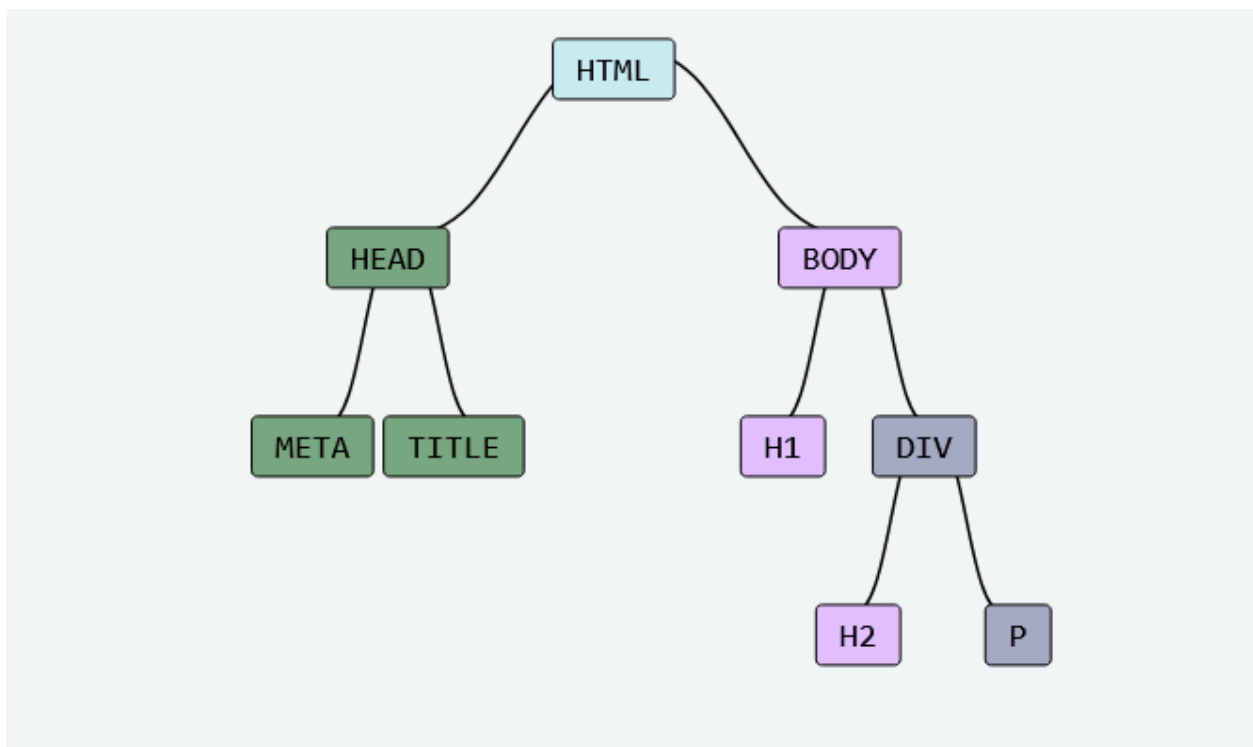
It is a representation of the HTML structure in a tree of objects where each part of your HTML document (elements, attributes, text) is represented as a node with parent-child relationships based on their nesting in the HTML code. It allows JavaScript and CSS styles to interact and manipulate the elements on the webpage.

Consider this HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>DOM Visualizer</title>
  </head>
```

```
<body>
  <h1>DOM Visualizer</h1>
  <div>
    <!-- This is a comment -->
    <h2>The StackSmith</h2>
    <p>Engineer / Developer</p>
  </div>
</body>
</html>
```

The DOM tree for the HTML code will look like this



[DIY here](#)

Why is DOM used?

Web browsers use the DOM to represent and manipulate HTML documents. For browsers, the tree-like structure enables easy identification and positioning of the HTML elements.

However, due to the implementation of a tree structure, any changes made to the DOM are resource-intensive and time-consuming. When DOM is altered the browser has to recalculate the element's size and position again and has to repaint the screen.

Here comes Virtual DOM, an efficient way to update the DOM. React's efficient update mechanism uses this virtual DOM to **only change the parts of the webpage** that need updating, instead of updating the whole page. This is what makes React Fast.

Understanding Virtual DOM

As the name suggests virtual DOM isn't a real DOM. It is represented as a tree structure using **JavaScript objects** and **is a representation of the actual DOM**. The virtual DOM is stored in memory and React uses virtual DOM to keep track of the changes and efficiently update the real DOM as needed.

For the example above, the virtual DOM will look like this

```
{
  tagName: "html",
  attributes: {},
  children: [
    {
      tagName: "head",
      attributes: {},
      children: [
        {
          tagName: "title",
          attributes: {},
          children: ["Virtual Document Object Model"],
        },
      ],
    },
    {
      tagName: "body",
      attributes: {},
      children: [
        { tagName: "h1", attributes: {}, children: ["DOM Visualizer"] },
        {
          tagName: "div",
          attributes: {},

```

```

      children: [
        { tagName: "h2", attributes: {}, children: ["The StackSmith"]
      },
        { tagName: "p", attributes: {}, children: ["Engineer"] },
      ],
    },
  ],
},
];
};

```

Fig: JavaScript Object Representation of DOM

Now let us take a JSX code and see how the virtual DOM looks like in React

```

import React, { useState } from 'react';
function App() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={() => setCount(count + 1)}>IncrementCounter</button>
    </div>
  );
}
export default App;

```

When you do

`console.log(App())`

the virtual DOM representation will look like this

```

{
  type: "div",
  $$typeof: Symbol(react.element),
  key: null,

```



```

ref: null,
props: {
  children: [
    {
      type: "h1",
      $$typeof: Symbol(react.element),
      key: null,
      ref: null,
      props: {
        children: [
          0: "Counter:",
          1: 0
        ],
      },
      _owner: null,
      _store: {},
    },
    {
      type: "button",
      $$typeof: Symbol(react.element),
      key: null,
      ref: null,
      props: {
        children: "IncrementCounter",
        onClick: () => setCount(count + 1)
      },
      _owner: null,
      _store: {},
    },
  ],
  _owner: null,
  _store: {},
};

```

And now when the Increase button is clicked once, only the h1 element is changed.

```
{
```

```

    type: "h1",
    $$typeof: Symbol(react.element),
    key: null,
    ref: null,
    props: {
      children: [
        0: "Counter:",
        1: 1
      ],
    },
    _owner: null,
    _store: {},
  },

```

The above code implies that the JSX has been parsed into a React Element or simply into a JavaScript Object.

***ReactElement* is a representation of a DOM element in the Virtual DOM.** [More About React Elements]

So how is the above virtual DOM created?

First of all, React starts to examine the JSX code recursively starting from the parent element. In this example, from the “div” element. The “div” has two children “<h1>” and “<button>”, and these children are specified in the “props” property of each element.

The <h1> element contains the text “Counter: 0”, which is treated as the child of the current element and thus is inside the “children” property of the “props” property.

```

{
  type: "h1",
  $$typeof: Symbol(react.element),
  key: null,
  ref: null,
  props: {
    children: [
      0: "Counter:",
      1: 0
    ],
  },
},

```

```

    _owner: null,
    _store: {},
  },
}

```

The `<button>` element has “IncrementCounter” text and an “onClick” event handler, which are also inside the “props” property.

```

{
  type: "button",
  $$typeof: Symbol(react.element),
  key: null,
  ref: null,
  props: {
    children: "IncrementCounter",
    onClick: () => setCount(count + 1)
  },
  _owner: null,
  _store: {},
},
}

```

So as you can see every nested HTML (DOM) element is translated as a child of the parent element, which creates a **tree** structure.

This is how DOM is translated into virtual DOM by React.

Understanding React Element fields

A React Element consists of many fields. but our interest will be in the following:

- **\$\$typeof:** This field having the value “**symbol.React**” is used to identify the react element in the virtual DOM. Any React element not containing this field may not be recognized as an element by React.
- **type:** The type specified in the virtual DOM generally corresponds to the type of HTML DOM element.
- **key:** The key property is the same key property we use while rendering a list of elements.
- **ref:** The ref property allows us to reference any HTML DOM element. If you have ever used `useRef`, you know about the ref property. More about refs in [React Documentation](#)

- **props:** This field contains the property values for your react component and its children. Event Handlers are also included inside this property.
 - **props.children:** The children field can accept both React elements and **null** values.
- So now you have a broad understanding of virtual DOM and React Elements.

How does Virtual DOM make React Faster?

Traversing through the DOM tree takes time complexity of $O(n)$, where **n** is the number of nodes. And after a change has been made in the DOM $O(n)$, our browsers need to further recalculate the element's size and position and then repaint the entire screen to load the changes.

As you can already have guessed by now, this isn't an efficient solution when we only need to change a small portion of the DOM like a text or something else.

Here comes Virtual DOM. As you already know, virtual DOM which is in memory is much faster and more efficient than the actual DOM, as updating a virtual DOM does not require heavy web browser processing like painting and recalibrating the space. So every time a React component updates, React constructs a new tree **in memory**.

Comparing and Updating the JavaScript objects in memory during a change is much faster than making changes to the actual DOM, making changes to an object has a time complexity of $O(1)$ which is much faster than the actual DOM's time complexity of $O(n)$. And not to mention we do not need to re-calibrate and repaint the entire screen. After changes have been made to the virtual DOM, React's efficient way of handling updates plays a crucial role in optimizing performance and enhancing the overall developer experience.

Reconciliation

Reconciliation is the key feature of React which efficiently updates the DOM by only making necessary changes in the DOM. After every component update, React creates a new virtual DOM tree and this newly created virtual DOM tree is compared with the old virtual DOM to identify the changes.

The reconciliation process involves the following key steps:

1. Initially, React creates a virtual DOM which is a lightweight copy of the real DOM.
2. After every change made in either component, state, or props, React then creates a new virtual DOM with the new data.

3. React then uses a **diffing algorithm** that traverses through the virtual DOM tree to compare/analyze the changes made between the new virtual DOM and the previous one.
4. These steps help React determine which component needs to be re-rendered on the real DOM.
5. Finally, the changes are then made to the actual/real DOM.

This process of creating a new tree and then comparing it with the old tree and then updating the UI is known as **Reconciliation**.

Even during multiple state changes, due to the reconciliation process, the real DOM is updated only once ensuring efficient rendering and a smooth user experience.

Diffing Algorithm

React needs to compare the changes between the old and new virtual DOM trees and also needs to ensure that a minimal number of changes are made to the actual/real DOM.

This process of efficiently handling the changes by comparing the two trees is done by Diffing Algorithm and it's the heart of the Reconciliation process.

This algorithm is quite complex but it is based on these 3 key assumptions:

1. Two elements of different types will produce different trees.

When the root elements of the Virtual DOM trees have different types, React's diffing algorithm discards the entire old DOM tree and constructs a new one from scratch. Consider this example:

```
<nav>
  <Header />
</nav>
```

If the JSX changes to

```
<div>
  <Header />
</div>
```

In this example, the root node transforms from `<nav>` to `<div>`. Since the root node has been changed, the diffing algorithm unmounts and destroys all the states of the old Component and then subsequently reconstructs the tree again.

This behavior is based on the assumption that elements of different types will produce different Virtual DOM structures.

Therefore, when the root element's type changes, the diffing algorithm assumes that the entire subtree has been replaced, thus, requiring a complete rebuild which ensures that the DOM remains in sync with the application's state.

2. Elements Of The Same Type

Let us see how the diffing algorithm behaves with DOM Elements and Component Elements of the same type.

a. DOM Elements Of The Same Type

When comparing two DOM Elements of the same type, React only looks at the attributes and keeps the same DOM node making updates to only the changed attributes. Consider this example:

Original DOM:

```
<div className="before" title="title" />
```

Updated DOM:

```
<div className="after" title="title" />
```

Here, the root element i.e. `<div>` remains the same so the **diffing algorithm efficiently only updates the className attribute** on the existing DOM element, preserving its underlying structure.

b. Component Elements Of The Same Type

This is the same as the updating DOM Element of the same type as described above but instead of the attributes in DOM Element, the Component Element focuses on updating the properties and state of the existing Component.

Consider this example:

```
<Counter count={0}>
  <button onClick={() => setCount(count + 1)}>+</button>
  <button onClick={() => setCount(count - 1)}>-</button>
  <span>{count}</span>
</Counter>
```

Consider we have a React component **Counter** that displays the counter value and provides buttons to increment and decrement the count value.

Now, when the user clicks on the increment button, the counter value updates to 1. Behind the scenes, React's diffing algorithm will compare the old and new Virtual DOM trees and since the root element (**Counter**) is of the same type, the diffing algorithm **will focus on updating the properties, and state of the existing component instance** and will not create a new DOM element from scratch.

3. Recursing on Children

While recursing on the children of a DOM node, React iterates over both lists of children at the same time and generates a mutation whenever there's a difference.

Consider this example

Original DOM:

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>
```

Updated DOM:

```
<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

In this example, the original DOM contains two elements within the `` list. But in the updated DOM, a new element, `third`, is added to the end of the list.

The diffing algorithm straightforwardly compares the tree structures and identifies the new element. Then it seamlessly adds the third element to the DOM. This process is clear and intuitive.

But let us consider this example

Original DOM:

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>
```

Updated DOM:

```
<ul>
  <li>zero</li>
  <li>first</li>
  <li>second</li>
</ul>
```

Now in this example, in the updated DOM a new element is added at the first position in the list. As a result, when the diffing algorithm compares the children the first element in the list has been changed, i.e., `first`, and `zero`.

Because the first elements are different, the diffing algorithm reconstructs the first element. Similarly, when comparing the second element, `second`, and `first`, it again needs to reconstruct the element. This approach results in the entire list being reconstructed, which is not an efficient process at all.

To overcome this issue, the keys are introduced in React.

We can hint at which child elements may be stable across different renders using a key prop.

Original DOM:

```
<ul>
  <li key="first">first</li>
  <li key="second">second</li>
</ul>
```

Updated DOM:

```
<ul>
  <li key="zero">zero</li>
  <li key="first">first</li>
  <li key="second">second</li>
</ul>
```

Now the Diffing algorithm can efficiently compare the keys and only update those elements whose keys have been modified. This approach results in a more efficient update of the DOM, thus optimizing the rendering process.

That's the reason we see a warning to provide keys to list items that React throws.

The mistake many people make and which I also have been guilty of doing for a very long is providing the index of the array as a key.

```
list.map((item, index) => (
  <li key={index}>{item}</li>
));
```

This is not a good practice as it may cause the displaying of wrong data or performance issues. Why?

Due to Array's unstable Identity

Array indices are inherently unstable i.e. when items are added, removed, or reordered within an array, the indices of the remaining items shift.

Suppose we add an item to the middle let's say in the 4th position of the list. If we use the index as the key. Then during comparison, the key is not changed i.e. the indexes always start from 0 and go on incrementally.

So can you guess what happens now?

Since the keys (indexes) are the same as before, React assumes that the DOM elements represent the same component as before. While traversing, only after the nth element, React finds a new key and only then React knows a new element has been added so React adds the new item to the list i.e. in the end.

Using the index as a key can confuse React's reconciliation algorithm. In the above example, React needed to update the 4th element, but instead updated the last element. Why? Because the array's indices had shifted and the only changed index was the nth index which caused incorrect updates and introduced unpredictability in our application.

Types of Reconciliation Algorithm

1. Stack reconciliation algorithm:

Before React 16, the diffing algorithm was responsible for identifying changes between the Virtual DOM and the actual DOM, and the **Stack reconciliation algorithm was tasked with applying those changes to the DOM**. This means that every change that needed to be updated was pushed onto the stack and then executed synchronously.

This introduced various challenges:

1. The Stack Reconciliation Algorithm could only process one update at a time, leading to delays when multiple state changes occur simultaneously.
2. When transitioning from state A to C in a sequence like $A \rightarrow B \rightarrow C$, the ideal scenario would be to directly display the eventual state C on the screen. However, in a stack-based algorithm, the intermediate state B is visible between A and C. This introduced inefficiencies and also the transition was visible on the screen.
3. Also, while these updates were being made, the UI was unresponsive.
4. It also affected the performance in the case of animations.

2. React Fiber reconciliation algorithm:

To address the limitations of the stack-based reconciliation, from React 16 onwards the Fiber reconciliation algorithm, **a more performant and flexible approach to updating the DOM** was introduced.

The various improvements over the Stack-based approach include:

1. React Fiber can handle multiple updates simultaneously, allowing the UI to remain responsive even when multiple state changes occur.
2. React Fiber employs a lane-based scheduling process to **prioritize updates based on their urgency**, ensuring that critical updates are processed promptly while non-essential updates can be done afterward.
3. React Fiber breaks down the updates into smaller chunks. This allows the algorithm to work with other tasks as well. Tasks such as handling user interactions or rendering animations. This ensures the UI is always responsive even when updates are being applied.

[Learn more about React Fiber Architecture](#)

Higher Order Components

Hooks

- useState
- useEffect
- useRef
- useMemo
- useCallback
- useReducer

Optimizing React Applications

React Under the Hood

React is a free and open-source front-end JavaScript library that aims to make building user interfaces based on components more "seamless", [Wikipedia](https://en.wikipedia.org/wiki/React_(software)). React is getting more and more popular nowadays and almost all the companies that plan on building a groundbreaking front-end system have surely given a thought if they should be using React for their next groundbreaking app.

When I started building React Applications myself, I had never really paid attention to how React works under the hood, and I never really understood what the most common terms meant, i.e. Virtual Dom, LifeCycle of Components, DOM Updates. These weren't a concern for me at all as I just wanted to build my app fast. I started my journey working with functional components, and all I could care about were the things that could help me build faster i.e. the hooks; for starters, all I cared about was how to use "props", "useState" and "useEffect". After all, these were the only things I needed to build my application and ship them to production.

pre-wrap;">But if you have ever deployed a React application, you might have already pointed out several issues that will arise due to the lack of fundamental knowledge. It would be just like building a car, just because you know the tire has to move and it needs gas and then you immediately jump to start building the car, and now viola! You need a fuel pump because your initial assumption of just pouring the fuel into the engine and hoping it would work has backfired, you don't even know how to control the timing of the engine piston. Someway somehow you made the car run for a few kilometers just by chance and now the car can't make a turn because you don't know anything about a differential. You see the point I am trying to make. So now let's get started.</p>