REACT JS

ALPHA TO OMEGA

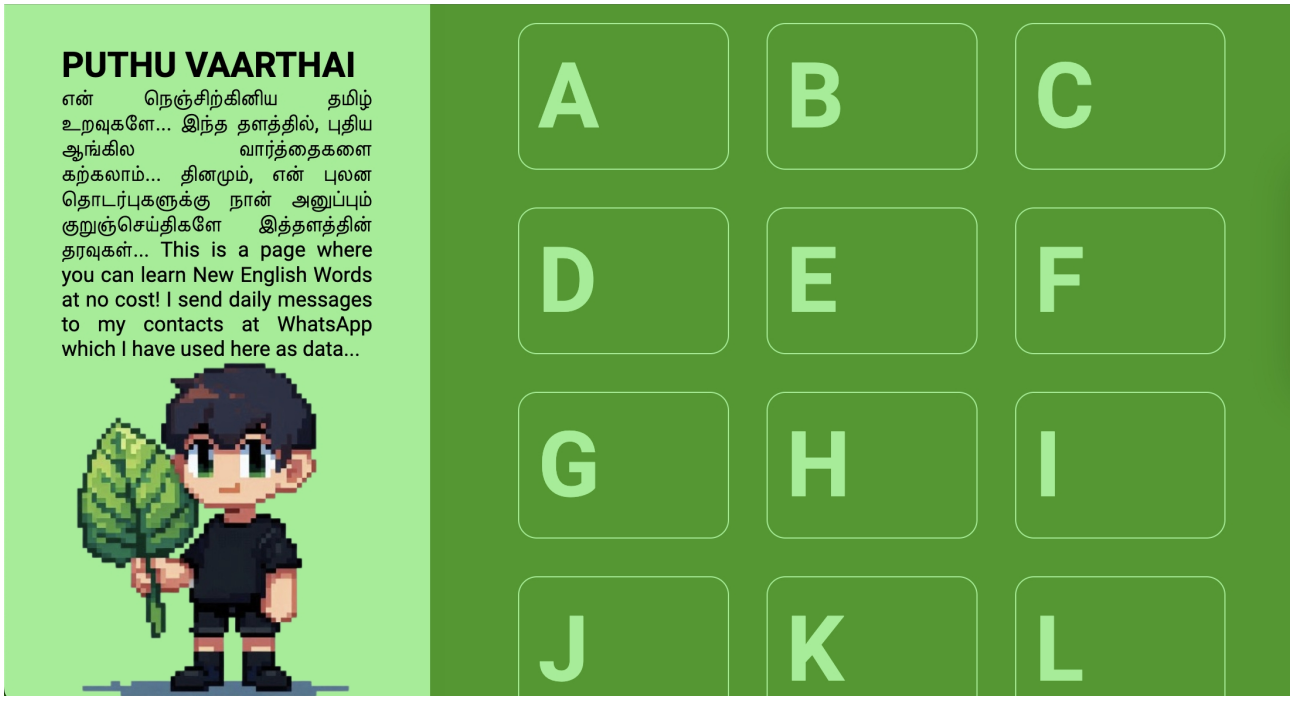
THIS BOOK

This book is written (typed) by Ari, who hails from the South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website, where you can reach out to him.

https://arihara-sudhan.github.io

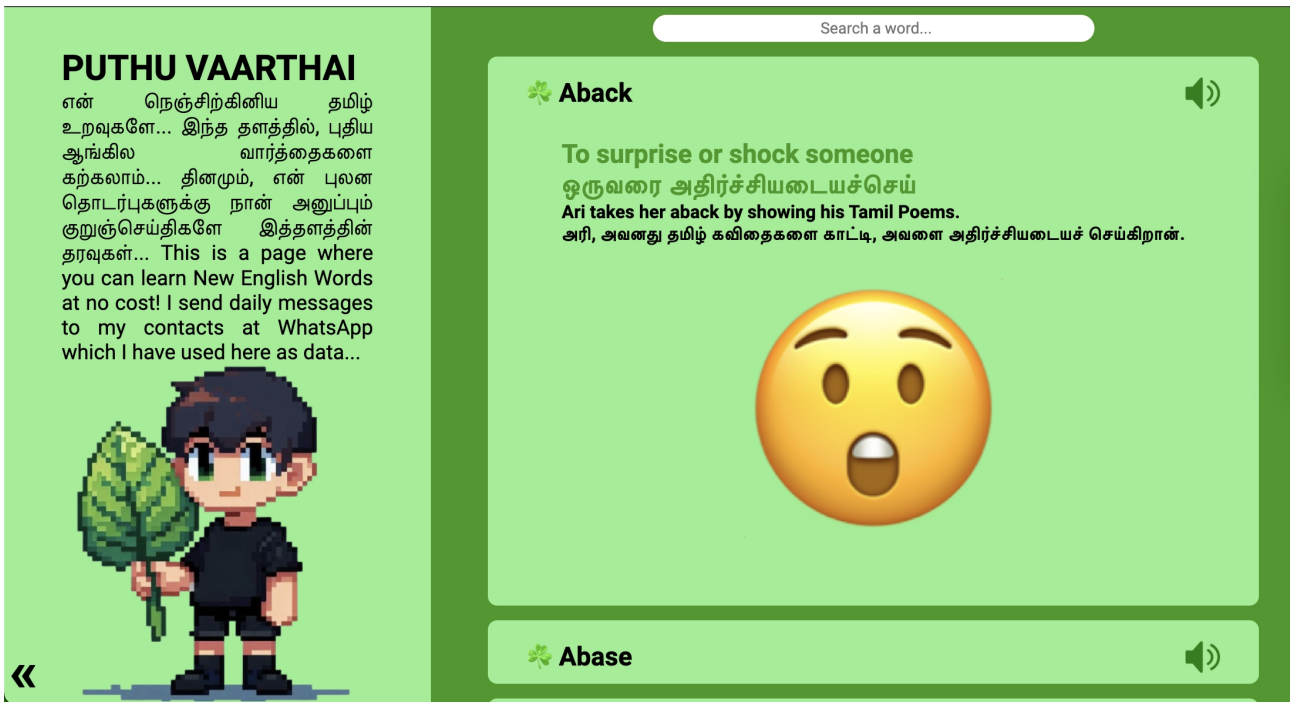
☆ World of React

ReactJS is a JavaScript library used for building user interfaces. It was developed by Facebook (Meta) and is widely used for creating single-page applications (SPAs), where the entire application is loaded once, and the content is dynamically updated without refreshing the page. Following is an example for a Single Page Application.



It is a web page made with React. On click a letter, it won't refresh the page but take as to a changed page.

The page never reloads!



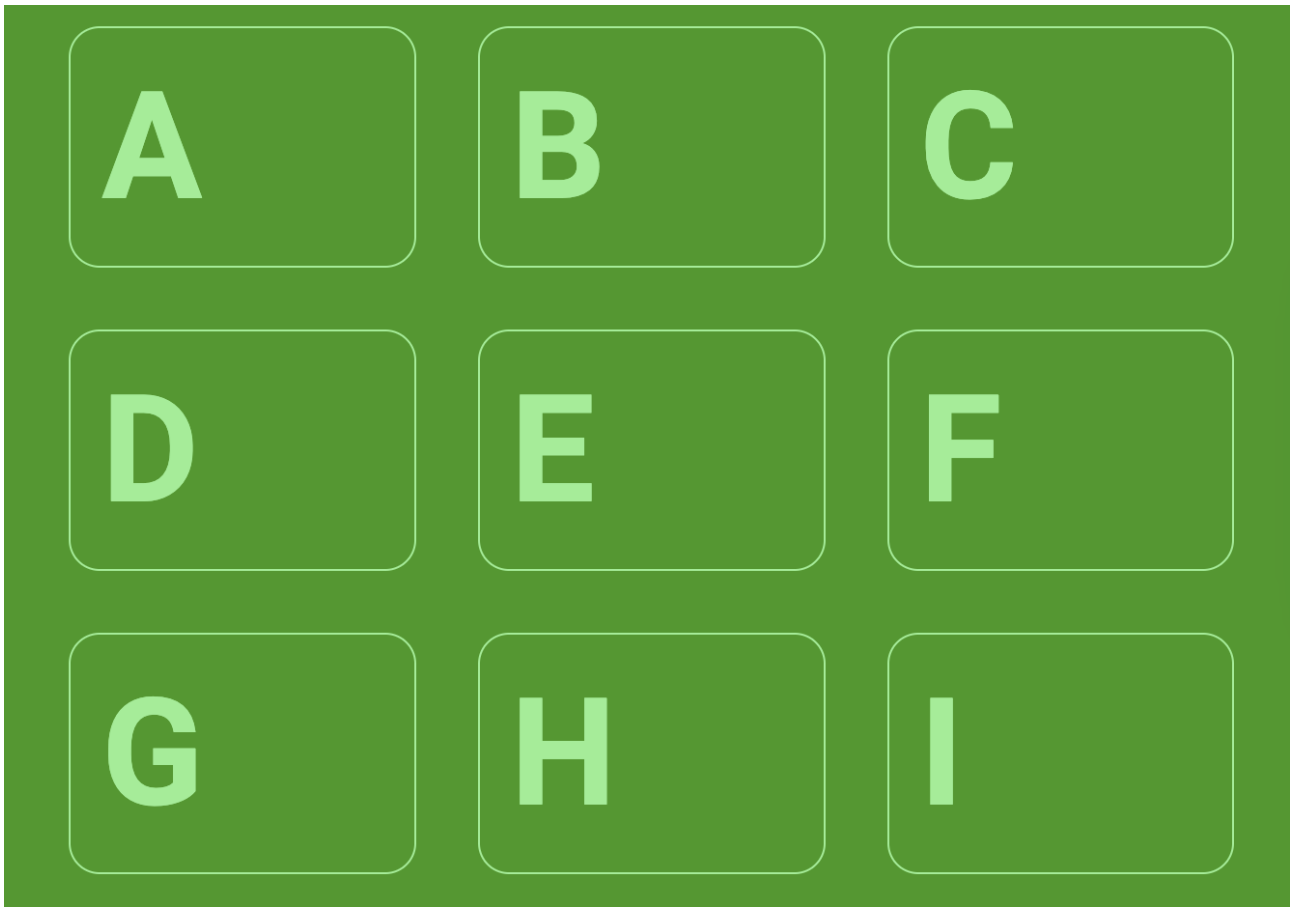
You can visit here. React breaks down a user interface into reusable, components. Each component is responsible for rendering a part of the UI and managing its own state.

☆ Components

Following are the separate components of our page.

Component 1: LeftBar

Component 2: RightBar



Component 3: Vocab

Modular components can be reused throughout an application or across multiple projects. Once a component is created, it can be reused wherever its functionality is needed, which reduces duplication of code and speeds up development. Bugs and issues are easier to isolate because they are typically confined to a specific component.

React's component-based architecture promotes a clear separation of concerns. Each component is responsible for one piece of the UI and handles both its own rendering logic and, optionally, state. Developers can collaborate more easily. Since components are self-contained, multiple developers can work on different components simultaneously.

☆ Reflow and Repaint

In native web systems, the actual DOM is manipulated directly whenever there is a change in the UI. This means that each time a change is made, whether it's adding, modifying, or removing elements, the browser, recalculates the layout of the page, repaints the content (if necessary). Manipulating the DOM directly can trigger layout recalculations and screen

redraws (called repaints). These operations can be quite expensive, especially if many changes are happening frequently or across many DOM elements. Accessing and manipulating the DOM frequently can be slow because the DOM is not optimized for rapid changes. Each change requires interaction with the browser's rendering

engine, which is a costly process.

Browser		Javascript Engine	
Chrome			**V8 Engine** (OpenSource)
Firefox			**Spider Monkey** (OpenSource)
Safari			**JavaScriptCore** (OpenSource)
Internet Explorer			**Chakra** (Proprietary)

Interacting with the browser's rendering engine is necessary because it handles converting the DOM into what users visually see. If we're not careful, native DOM manipulation can cause inefficient updates. For example, making several changes to the DOM individually can lead to performance bottlenecks.

React’s DOM update process is different from traditional DOM manipulation in how it minimizes the number of direct updates to the actual DOM. This is achieved through the use of the Virtual DOM, which allows React to optimize performance by making efficient updates, rather than triggering costly reflows and repaints for every change.

React uses an in-memory Virtual DOM, which is a lightweight copy of the actual DOM. When a component's state or props change, React updates the Virtual DOM first, rather than the real DOM. This enables React to batch multiple updates and compare the changes more efficiently before applying them to the actual DOM.

Whenever a component's state or props change, React automatically re-renders the component by updating the Virtual DOM. This step happens entirely in memory, so it's much faster than working with the actual DOM. React uses a process called reconciliation to determine what has changed between the previous Virtual DOM and the new Virtual DOM.

React compares the two versions using a diffing algorithm, which identifies only the specific parts of the DOM that have changed. Instead of updating the entire page or entire component, React narrows down the changes to the specific elements or attributes that need to be updated. This results in fewer and more targeted DOM operations.

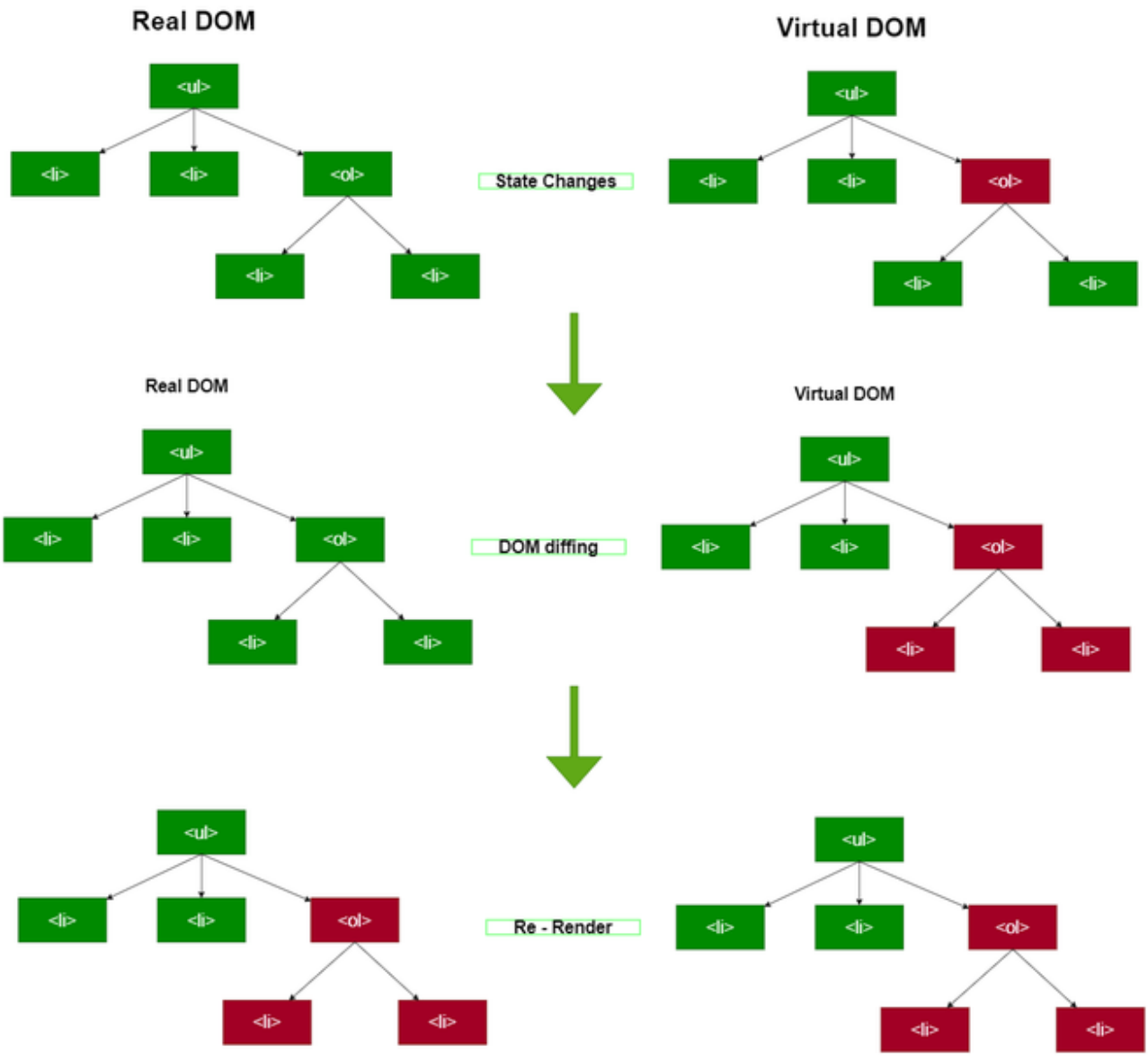


Image Courtesy: GeeksforGeeks

React batches updates in a way that multiple state changes or DOM updates are processed together, rather than one at a time.

This further reduces the number of direct DOM manipulations, as React can group changes and make them all at once, avoiding multiple reflows and repaints. Once React has determined the minimal set of changes needed (based on the diffing process), it updates the actual DOM with just those changes.

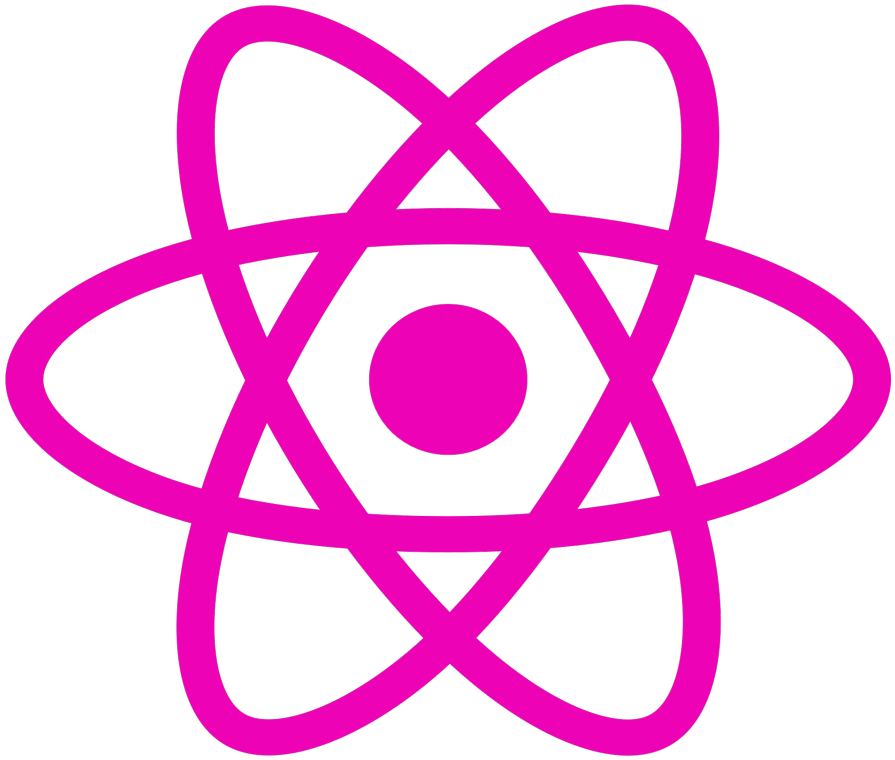
This ensures that the browser’s rendering engine only recalculates the layout or repaints for the specific elements that were affected, reducing performance overhead. So, the two phases are, Render Phase (or "Reconciliation Phase") and Commit Phase.

In Render phase, React updates the Virtual DOM in memory. When a component’s state or props change, React recalculates

the component's new Virtual DOM representation by rendering the component again. React then uses its diffing algorithm to compare the new Virtual DOM tree with the previous one, figuring out the smallest set of changes that need to be applied to the actual DOM. The render phase is purely in memory.

It doesn’t interact with the actual DOM yet, so no reflows, repaints, or any browser rendering tasks happen during this phase. Once the render phase is complete, React moves to the commit phase, where it applies the calculated changes from the Virtual DOM to the actual DOM. React batches the updates to the actual DOM and then commits the changes all at once.

This minimizes the number of interactions with the real DOM, which helps improve performance. During this phase, React interacts with the browser's rendering engine, triggering layout recalculations, reflows, and repaints as necessary based on the changes.



☆ REACT ABSTRACTS

React allows developers to describe what they want the UI to look like based on the current state, rather than managing the UI updates manually. React then takes care of the rest.

We describe what the UI should look like based on the current state, and React takes care of updating the actual DOM to match that description.

If it is Vanilla JS, there will be some imperative style in the code.

MANUALLY DO EVERYTHING

```
let count = 0;
const counterElement = document.getElementById('counter');
const buttonElement = document.getElementById('increment-btn');
buttonElement.addEventListener('click', function () {
count++;
counterElement.textContent = 'Counter: ' + count;
});
```

Now, if it is React, it will be like a description.

RELY ON REACT

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

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Counter: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
export default Counter;
```

☆ REACT IS A LIBRARY

React is not a framework. Most people prefer React because it gives more flexibility. React lets us piece together our app however we want. An opinionated structure is not always loved. React didn't impose any constraints on us. That's worth something, right? So, look at the following picture.

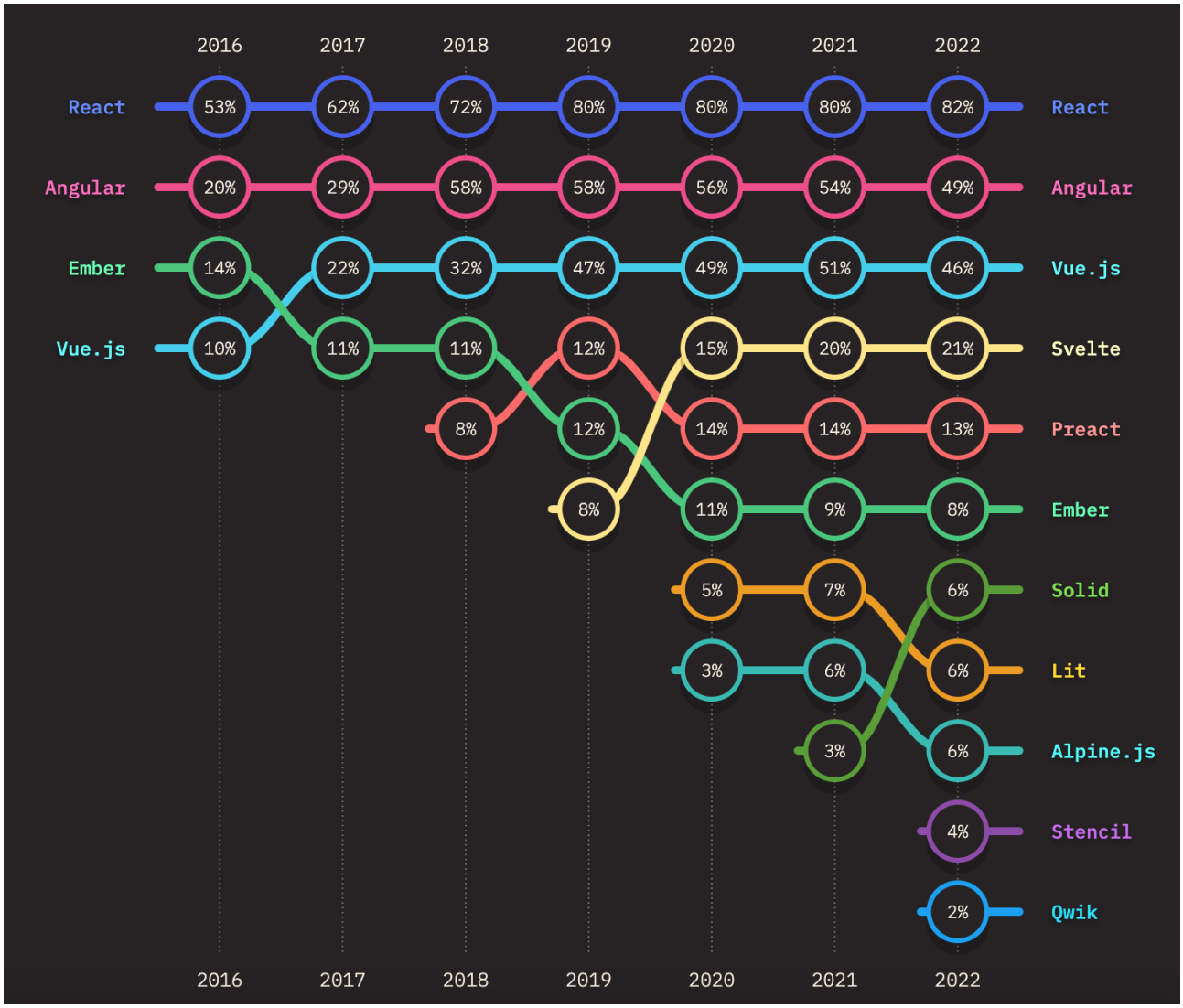


Image Courtesy: Pretius

☆ ECOSYSTEM

There are thousands of packages we probably don't need! **#JFF**

☆ INSTALL

React requires Node.js and npm (Node Package Manager) to be installed on our machine. Download the latest version of Node.js from **https://nodejs.org** and install it. The easiest and recommended way to set up a React project is, using create-react-app. It automates the setup, so we don't have to manually configure things.

Run the following command to install create-react-app globally (if you haven't done so already).

> npm install -g create-react-app

Create a new React project by running,

Replace "my-app" with the desired name for your project.

> npx create-react-app my-app

You will see a strange message of “Happy Hacking”. Now, Navigate to your project folder.

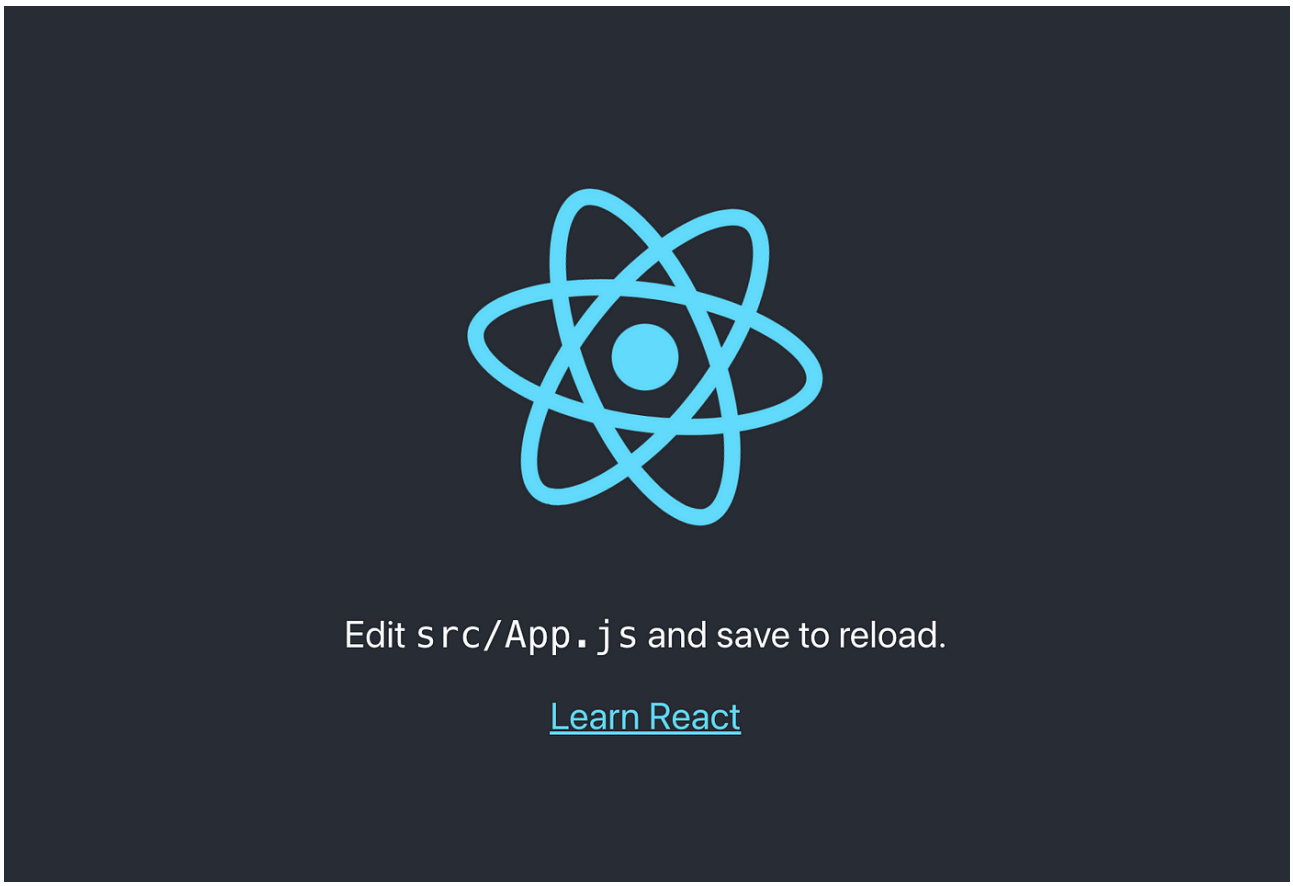
> cd my-app

Start the development server.

> npm start

This will open our application in the browser at http://localhost:3000.

You will probably see a logo of rotating atom.



We are ready now!

When we create a React application using create-react-app, it generates a basic folder structure for us. This structure is flexible,

and we can reorganize it as our project grows, but it provides a solid starting point for small to medium projects.

```
my-app/
├──── node_modules/
├──── public/
|     ├──── favicon.ico
|     ├──── index.html
|     ├──── manifest.json
├──── src/
|     ├──── App.css
|     ├──── App.js
|     ├──── App.test.js
|     ├──── index.css
|     ├──── index.js
|     ├──── logo.svg
├──── .gitignore
├──── package.json
├──── README.md
├──── yarn.lock or package-lock.json
```

node_modules/

This folder contains all the installed dependencies (packages) our project needs. It is created when we run npm install or yarn install and stores third-party libraries. we typically don't touch this folder directly.

public/

This folder contains static files that are publicly accessible. Here resides the index.html: The only HTML file in the project.

React injects our app into the tag inside this file. It's the entry point for our app. The icon displayed in the browser tab, favicon.ico is also here. Anything placed in the public folder will be accessible by URL. For example, if we place an image there, we can access it by navigating to http://localhost:3000/img.png in development.

src/

It contains all the source code for our React application. index.js is the entry point for our React application. It renders the root component (usually <App />) into the DOM. App.js is the main component of our React app. It's where the content and components of our application start. We can break down this file into smaller components as our app grows. Some css, test files and logo will be there.

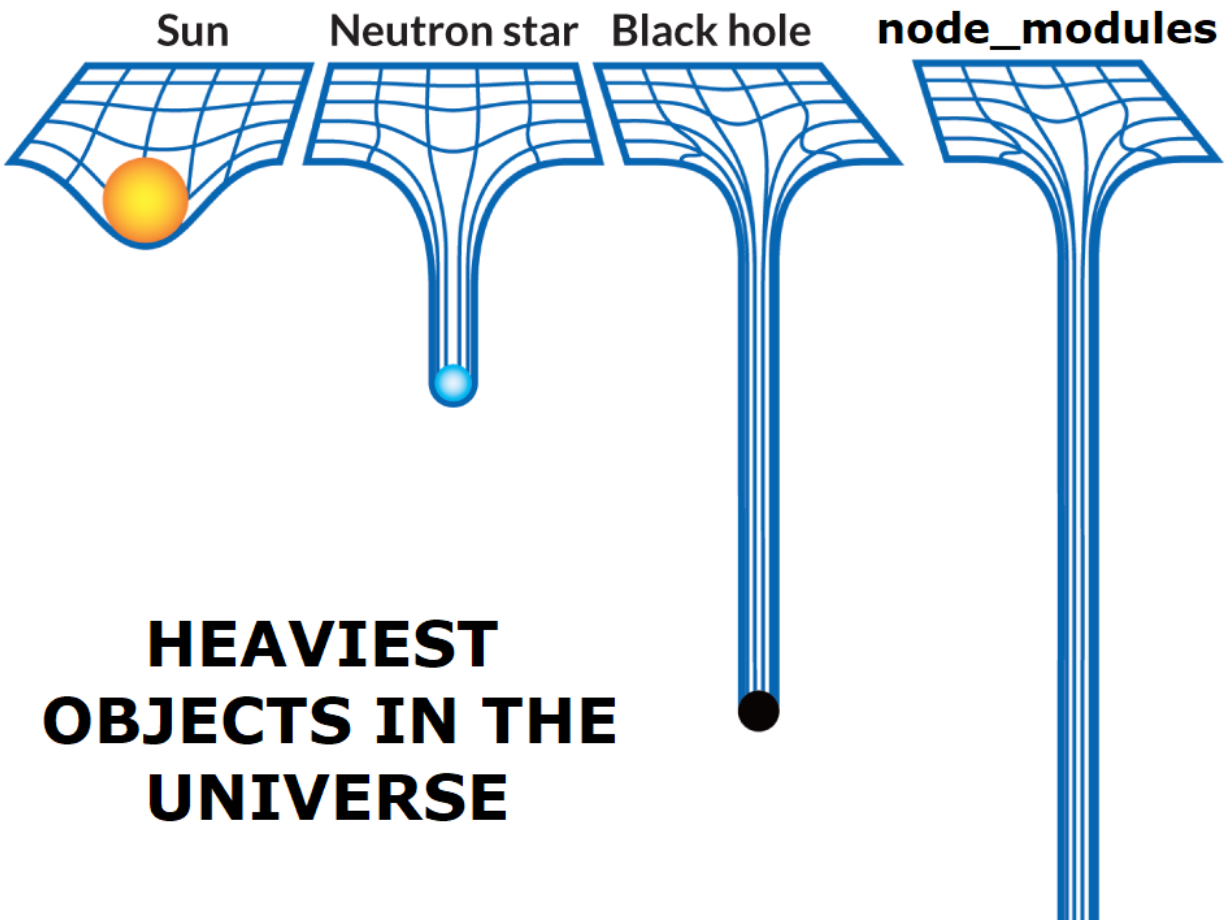
package.json

This file contains metadata about the project and lists the dependencies, scripts, and configurations. It lists all the libraries and tools our project depends on. The scripts section specifies npm/yarn scripts that can be run from the command line.

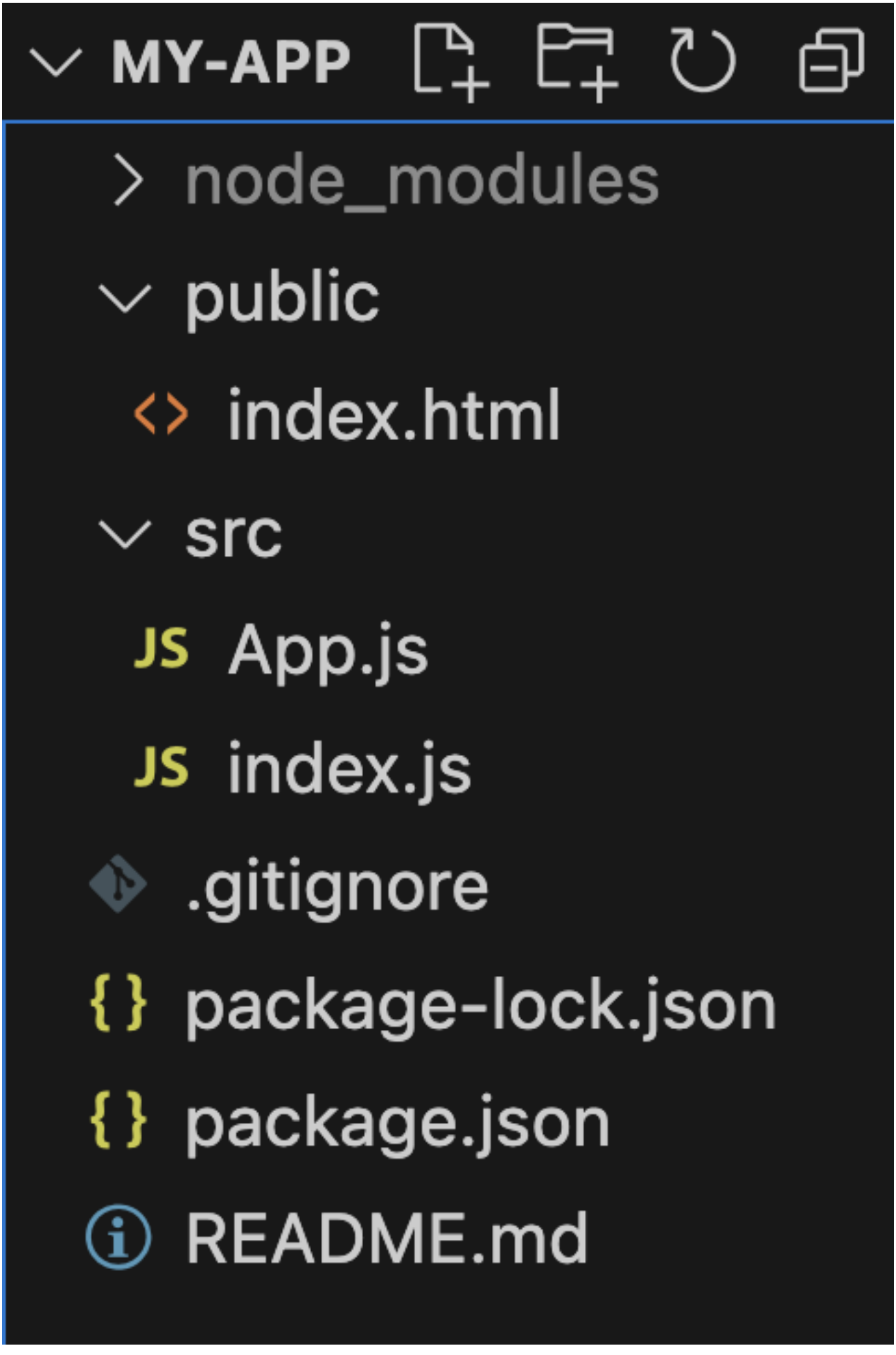
package-lock.json

This file locks the exact versions of our project's dependencies.

It ensures that every team member and the production environment installs the exact same versions of the dependencies, avoiding version conflicts. If we lost all our modules in node_modules/, we can simply install them back by npm install command as long as we have this lock file. Look at the following memes for even more good intuitions.



So, let's clean our project to look like the following. Remove the lines where you use the deleted dependencies.


MY-APP
node_modules
public
index.html
src
App.js
index.js
.gitignore
package-lock.json
package.json
README.md

☆ CLASS COMPONENTS

The class components are the primal way of defining components in React. It needs a class to extend React.Component and to keep a render method inside. Let's write something in our App.js.

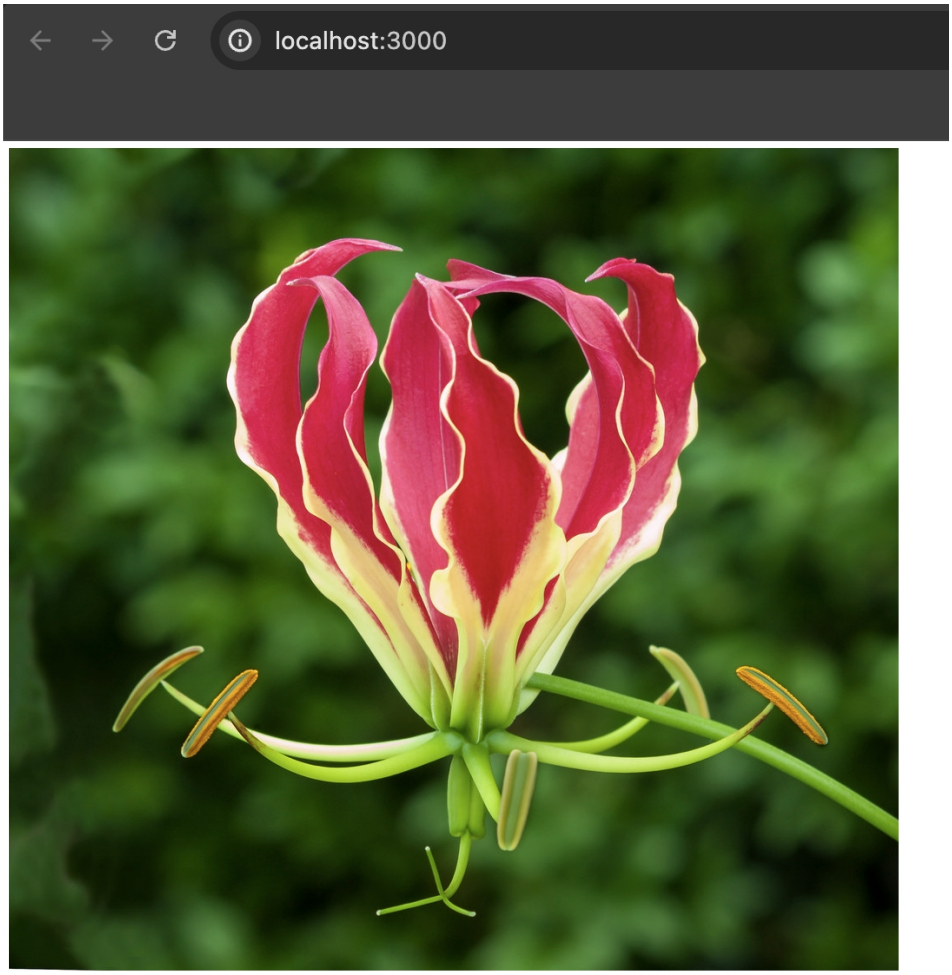
My index.js is also cleaned like the following.

src > JS index.js > ...

You, 2 weeks ago | 1 author (You)

```
1  import React from 'react';
2  import ReactDOM from 'react-dom/client';
3  import App from './App';
4
5  const root = ReactDOM.createRoot(document.getElementById('root'));
6  root.render(
7    <React.StrictMode>
8      <App />
9    </React.StrictMode>
10 );
```

Now, let's look at the output in localhost.



🌸 **Glory Lily blossoms!**

It is the followed way in class components. To make it stateful, we can have a state object there in constructor. The constructor is the ideal place to initialize this.state, which defines the initial state of the component.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Lily",
      scname: "gloriosa superba"
    }
  }
  render() {
    return <div>
             <img src="/images/gloriosa.jpg" alt="An Error Occured!"/>
             <h1>NAME: {this.state.name}</h1>
             <h1>SCIENTIFIC NAME: {this.state.scname}</h1>
           </div>
  }
}
```

To access any state properties, we have to use this.state. Ensure to export the component so that it can be imported in index.js.

If we are extending a class and using a constructor, we must call super() before we use this. In React, calling super(props) initializes this within the context of the component and passes props to the parent class's constructor.

To update the state, there is a method called setState available in this.

```
You, 28 seconds ago | 1 author (You)
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Lily",
      scname: "gloriosa superba"
    }
  }
  handleClick = () => {
    this.setState({name: "Glory Lily"})
  }
  render() {
    return <div>
            <img src="/images/gloriosa.jpg" alt="An Error Occured!"/>
            <h1 onClick={()=> {this.handleClick()}}>NAME: {this.state.name}</h1>
            <h1>SCIENTIFIC NAME: {this.state.scname}</h1>
           </div>
  }
}
```

Here, to access the methods available, we have to use this again!

On clicking the name, “Lily”, it will change to “Glory Lily”.

NAME: Lily

SCIENTIFIC NAME: gloriosa superba

NAME: Glory Lily

SCIENTIFIC NAME: gloriosa superba

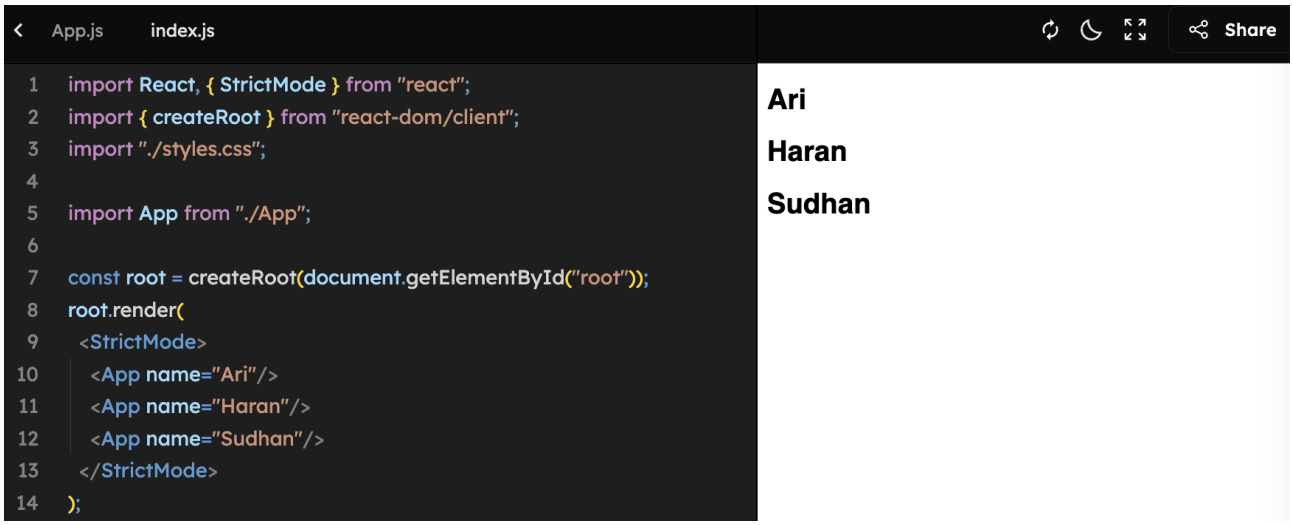
Another thing is props. As you can see there in the constructor part, we have talked of props. But, what is it? It is a way of sending data from parent to child component.

App.js index.js

```
1  import React from "react";
2  export default class App extends React.Component{
3      constructor(props) {
4          super(props);
5      }
6      render() {
7          return <h1>{this.props.name}</h1>
8      }
9  }
10
```

Here, we use props!

How will it help us? We access the prop name using this.props.name. How does it help us? It helps us in Reusability of code. In the index.js, let's render the component multiple times with different props value for name.



Props cannot be changed. The parent passes data to child through the so called props. The props.children is a special prop that allows you to pass elements (such as text, components, or other JSX) between the opening and closing tags of a component. It enables a component to act as a container or wrapper for whatever you put inside it.

```jsx
import React from "react";
export default class App extends React.Component{
    constructor(props) {
        super(props);
    }
    render() {
        return <div>{this.props.children}</div>
    }
}
```

In index.js, we will have the children wrapped by App.

```jsx
import React, { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import "./styles.css";

import App from "./App";

const root = createRoot(document.getElementById("root"));
root.render(
  <StrictMode>
    <App>
      <h1>HELLO</h1>
      <h1>HEY</h1>
    </App>
  </StrictMode>
);
```

HELLO

HEY

We can also have types for the props. It will log warning when there is a mismatch in props type.

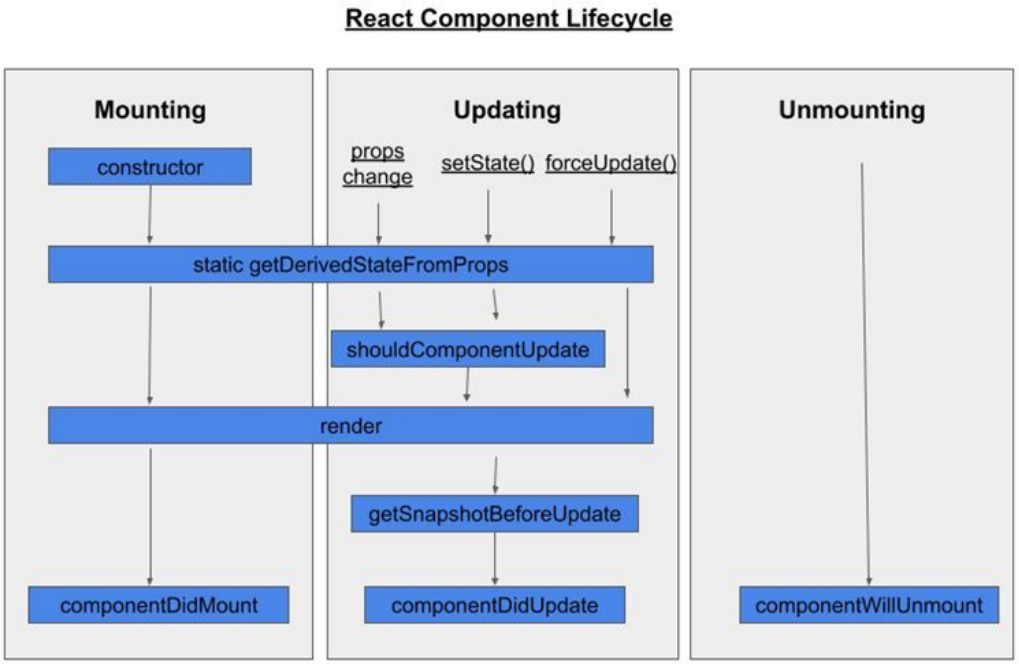
```
import React from "react";
import PropTypes from 'prop-types';

export default class App extends React.Component{
    constructor(props) {
        super(props);
    }
    render() {
        return <div>{this.props.name} is {this.props.age} years old!</div>
    }
}

App.propTypes = {
    name: PropTypes.string,
    age: PropTypes.number.isRequired
}
```

Note: propTypes will not cause the app to crash.

Lifecycle methods are special methods each component goes through during its life in an application.



These are mainly used in class components to perform actions at different stages.

1. MOUNTING

When a component is being created and inserted into the DOM, the following methods are called in order

> constructor(props)

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

```
1  import React from "react";
2
3  class App extends React.Component {
4    constructor(props) {
5      super(props);
6      this.state = {favoriteColor: "Black"};
7    }
8    render() {
9      return (
10       <h1>My Favorite Color is {this.state.favoriteColor}</h1>
11     );
12   }
13 }
14
15 export default App;
```

My Favorite Color is Black

> *getDerivedStateFromProps(props, state)*

The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM. This is the natural place to set the state object based on the initial props. It takes state as an argument, and returns an object with changes to the state.

```
1  import React from "react";
2
3  class App extends React.Component {
4    constructor(props) {
5      super(props);
6      this.state = {favoriteColor: "Red"};
7    }
8    static getDerivedStateFromProps(props, state) {
9      return {favoriteColor: props.favCol };
10   }
11   render() {
12     return (
13       <h1>My Favorite Color is {this.state.favoriteColor}</h1>
14     );
15   }
16 }
17 export default App;
```

My Favorite Color is Black

A Prop value is passed there in index.js.

```
root.render(
  <StrictMode>
    <App favCol="Black"/>
  </StrictMode>
```

> *render()*

The render() method is the method that actually outputs the HTML to the DOM.

```
1  import React from "react";
2
3  class App extends React.Component {
4    render() {
5      return (
6        <h1>My Favorite Color is Black</h1>
7      );
8    }
9  }
10 export default App;
```

My Favorite Color is Black

> *componentDidMount*

This method is called after the component is rendered. This is where we run statements that requires that the component is already placed in the DOM.

```
1  import React from "react";
2
3  class App extends React.Component {
4    constructor(props) {
5      super(props);
6      this.state = {favoriteColor: "Red"};
7    }
8    componentDidMount() {
9      setTimeout(() => {
10       this.setState({favoriteColor: "Black"})
11     }, 1000)
12   }
13   render() {
14     return (
15       <h1>My Favorite Color is {this.state.favoriteColor}</h1>
16     );
17   }
18 }
```

My Favorite Color is Black

2. UPDATING

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

> getDerivedStateFromProps(props, state)

Also at updates, this method is called.

```
3
4  class App extends React.Component {
5    constructor(props) {
6      super(props);
7      this.state = { favoriteColor: "Red" };
8    }
9    static getDerivedStateFromProps(props, state) {
10       return { favoriteColor: props.favCol };
11   }
12   render() {
13     return (
14       <div>
15         <h1>My Favorite Color is {this.state.favoriteColor}</h1>
16       </div>
17     );
18   }
19 }
```

My Favorite Color is Yellow

> *shouldComponentUpdate()*

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

```
3  class App extends React.Component {
4    constructor(props) {
5      super(props);
6      this.state = {favoritecolor: "red"};
7    }
8    shouldComponentUpdate() {
9      return true;
10   }
11   changeColor = () => {
12     this.setState({favoritecolor: "blue"});
13   }
14   render() {
15     return (
16       <div>
17       <h1>My Favorite Color is {this.state.favoritecolor}</h1>
18       <button type="button" onClick={this.changeColor}>Change color</button>
19       </div>
20     );
21   }
22 }
```

If the shouldComponentUpdate here returns false, the update cannot be proceeded.

> *render()*

Yes! It is also called when a component (state/props) gets updated.

> *getSnapshotBeforeUpdate()*

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

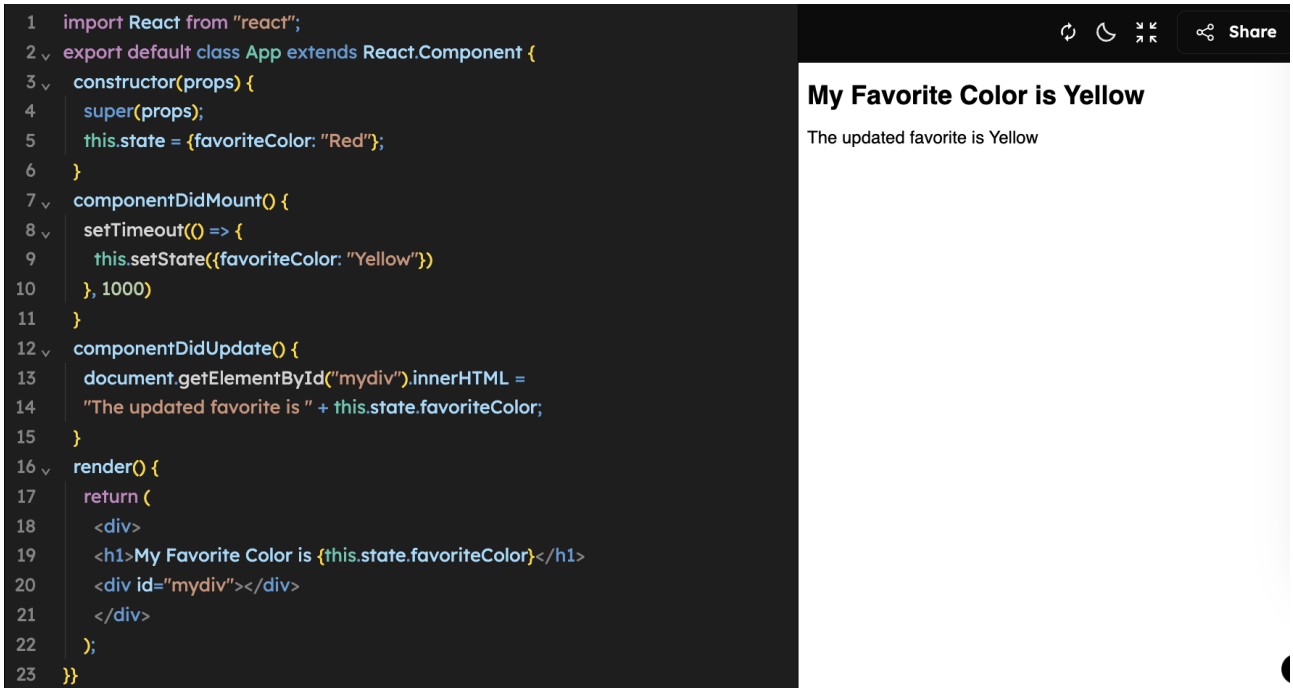
```
import React from "react";
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoriteColor: "red" };
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({ favoriteColor: "yellow" });
    }, 1000);
  }
  getSnapshotBeforeUpdate(prevProps,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

My Favorite Color is yellow

Before the update, the favorite was red
The updated favorite is yellow

> *componentDidUpdate()*

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



```
import React from "react";
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoriteColor: "Red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoriteColor: "Yellow"})
    }, 1000)
  }
  componentDidUpdate() {
    document.getElementById("mydiv").innerHTML =
    "The updated favorite is " + this.state.favoriteColor;
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoriteColor}</h1>
      <div id="mydiv"></div>
      </div>
    );
}}
```

Share

My Favorite Color is Yellow

The updated favorite is Yellow

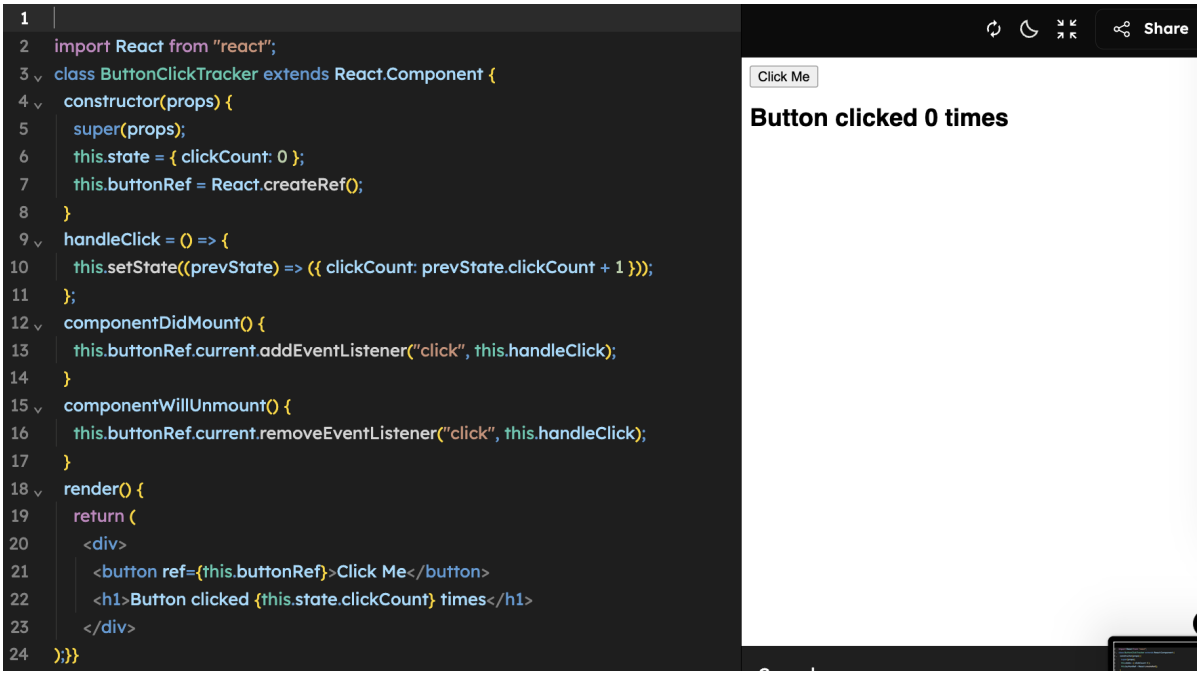
It is the apt place for API calling...

3. UNMOUNTING

This phase is when a component is removed from the DOM.

> *componentWillUnmount()*

The componentWillUnmount method is called when the component is about to be removed from the DOM. It is used to clean up resources just before a component is removed from the DOM.



☆ FUNCTION COMPOS

The another way of defining components is to use the functions of JavaScript. Such components are coined as Function Components.

```
2 ⌄ export default function App() {
3     return <h1>Hello world</h1>
4   }
```

Hello world

The props can be passed in the similar way we followed in class components. But here, we don’t have this.props.

```
import React, { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import "./styles.css";

import App from "./App";

const root = createRoot(document.getElementById("root"));
root.render(
  <StrictMode>
    <App name = "Ari" age={22}/>
  </StrictMode>
);
```

Now, we can access it using props.propName.

```
export default function App(props) {
  return <h1>I am {props.name}, {props.age} years old!</h1>
}
```

I am Ari, 22 years old!

As we have understood, the props are read-only. They are not state of a component. They are just a way of communication between Parent and Child Component. It is a one way communication

between Parent and Child.

If we want to make it look clean, we can use de-structuring of objects.

```
2  export default function App(props) {
3      const {name, age} = props;
4    return <h1>I am {name}, {age} years old!</h1>
5  }
```

Or never let the props in! De-structure it when it comes.

```
export default function App({name, age}) {
  return <h1>I am {name}, {age} years old!</h1>
}
```

To make it to have default props, we can simply assign values to the params (props).

```
export default function App({name="Ariharan", age}) {
  return <h1>I am {name}, {age} years old!</h1>
}
```

So, when the name props is not given from there, it will take the default one.

```
const root = createRoot(document.getElementById("root"));
root.render(
  <StrictMode>
    <App age={22}/>
  </StrictMode>
);
```

I am Ariharan, 22 years old!

Another way is to define the defaultProps object.

```
export default function App({name, age, native}) {
  return <h1>I am {name}, {age} years old! from {native}</h1>
}
App.defaultProps = {
    name: "Ariharan",
    age: 22,
    native: "Tirunelveli"
}
```

I am Ari, 22 years old! from Tirunelveli

Remember, we were using the destructured version of the props.

```
export default function App(props) {
  return <h1>I am {props.name}, {props.age} years old! from {props.native}</h1>
}
App.defaultProps = {
    name: "Ariharan",
    age: 22,
    native: "Tirunelveli"
}
```

Alright, another way of defining the components, is to use the arrow function notation as shown below.

```
const App = (props)  => {
  return <h1>I am {props.name}, {props.age} years old!</h1>
}
export default App;
```

Here is the small comparison b/w class and function components.

Aspect	Functional Components	Class Components
Definition	A simple JavaScript pure function that takes props and returns a React element (JSX).	Must extend from `React.Component` and define a `render()` method that returns a React element.
Render Method	No `render` method is used. JSX is returned directly from the function.	A `render()` method is required to return JSX, similar to HTML in syntax.
Execution	Executes from top to bottom. Once the function is returned, it's done with its execution.	Instantiates a class instance with lifecycle methods that can be invoked at different component phases.
State Management	Initially known as stateless, but can now manage state using hooks like `useState`.	Known as stateful components and manage their own state traditionally through `this.state`.
Lifecycle Methods	Cannot use lifecycle methods but can mimic them using hooks like `useEffect`.	Can use lifecycle methods such as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.
Hooks Usage	Implements hooks easily within the function body to handle state and effects.	Does not use hooks. State and lifecycle logic are handled differently through class methods.
Constructor	Does not require a constructor. Hooks handle state initialization and effects without it.	Requires a constructor for initializing state and binding event handlers.
Efficiency	More efficient due to less boilerplate and direct usage of hooks for state and effects.	Slightly less efficient. Can have more boilerplate code due to lifecycle methods and state management.
Code Complexity	Requires fewer lines of code, leading to simpler, more readable components.	Typically requires more code, which can lead to more complexity and verbosity.

The concept of propTypes in Function Components is just the same as in Class Components.

```
import PropTypes from 'prop-types';

const App = (props)  => {
  return <h1>I am {props.name}, {props.age} years old!</h1>
}

App.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number.isRequired,
};

export default App;
```

For components, we use PascalCase and for functions, we use camelCase.

Events are handled using props that correspond to the event type.

```
const App = (props)  => {
   const tellMyName = () => {
      alert("Ari");
   }
  return <>
      <button onClick={tellMyName}>TELL MY NAME</button>
    </>
}
```

To pass arguments to the event handlers, we have to wrap the event type call into another function. We shouldn’t directly call it.

```
const App = (props)  => {
    const tellMyName = (name) => {
        alert(name);
    }
  return <>
        <button onClick={()=> {tellMyName('Ari')}}>TELL MY NAME</button>
      </>
}
```

If we directly call the function, it will be called for each render, annoying.

We just have to mention the function reference over there in target event.

React provides a synthetic event as the first argument.

This synthetic event is a wrapper around the browser's native event, designed to normalize behaviour across different browsers.

```
const App = (props)  => {
   const tellMyName = (name, e) => {
      alert(e.target); // HTMLButtonElement
   }
  return <>
      <button onClick={(e)=> {tellMyName('Ari', e)}}>TELL MY NAME</button>
    </>
}
```

It is "BY DEFAULT".

```
const App = (props)  => {
   const tellMyName = (e) => {
      alert(e.target); // HTMLButtonElement
   }
  return <>
      <button onClick={tellMyName}>TELL MY NAME</button>
    </>
}
```

State is local to a component and allows a component to maintain its own data over time.

In Function Components, we have useState hook which return the passed

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

const App = (props)  => {
  const [age, setAge] = useState(21);
  const incrementAge = () => {
      setAge(age+1);
  }
  return <>
        <h1>Ari is {age} years old!</h1>
        <button onClick={incrementAge}>HBD</button>
      </>
}
```

state and a setter function.

No one else than this setter can change the state. The component re-renders every time the state changes. When you call the setAge function React marks that component as "dirty" and schedules a re-render.

Besides, we can update the state condtionally.

```
const incrementAge = () => {
    setAge(age%2==0 ? age+1 : age-1);
}
```

So, what is a hook? Hooks allow us to hook into the features of class components in functional components. I'm talking of the Life Cycle Methods.

We can also check the previous state before updating the current state.

```
const App = (props)  => {
  const [name, setName] = useState("Ari");

  const changeName = () => {
    setName(prevName => {
      return prevName === "Ari" ? "Aravind" : "Ariel";
    });
  }
```

```
  return (
    <>
      <h1>NAME: {name}</h1>
      <button onClick={changeName}>CHANGE NAME</button>
    </>
  );
}
```

Alright! Look at the following. We can handle complex states such as an object or an array in useState.

```
const App = (props)  => {
  const [array, setArray] = useState([]);
  return <>
      <button onClick={()=>{ setArray([1,2]) }}>MAKE ARRAY</button>
    </>
}
```

Beware of setting objects as there is a chance to lose data. Make sure to spread the previous object.

```jsx
const App = (props)  => {
  const [ari, setAri] = useState({name: "Ari", age: 21});

  const updateAri = () => {
    setAri(prevState => {
      return {...prevState, name: "Aravind"};
    });
  }

  return (
    <>
      <h1>NAME: {ari.name}</h1>
      <h1>AGE: {ari.age}</h1>
      <button onClick={updateAri}>SET ARI</button>
    </>
  );
}
```

So, here is the output.

NAME: Ari

AGE: 21

SET ARI

NAME: Aravind

AGE: 21

SET ARI

Age is preserved!

Map is another powerful thing in React. With less code, we achieve large.

```jsx
const App = () => {
  const agents = [
    { name: "Ari", age: 30 },
    { name: "Haran", age: 25 },
    { name: "Sudhan", age: 35 },
  ];

  return (
    <>
      {
        agents.map((agent, index) => {
          return (
            <div key={index}>
              <h1>{agent.name}</h1>
              <h2>{agent.age}</h2>
            </div>
          );
        })
      }
    </>
  );
};
```

We can conditionally render a component.

```jsx
const App = () => {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  return (
    <>
      {isLoggedIn ? (
        <div>
          <h1>Welcome back, User!</h1>
          <button onClick={() => setIsLoggedIn(false)}>Log out</button>
        </div>
      ) : (
        <div>
          <h1>Please log in</h1>
          <button onClick={() => setIsLoggedIn(true)}>Log in</button>
        </div>
      )}
    </>
  );
};
```

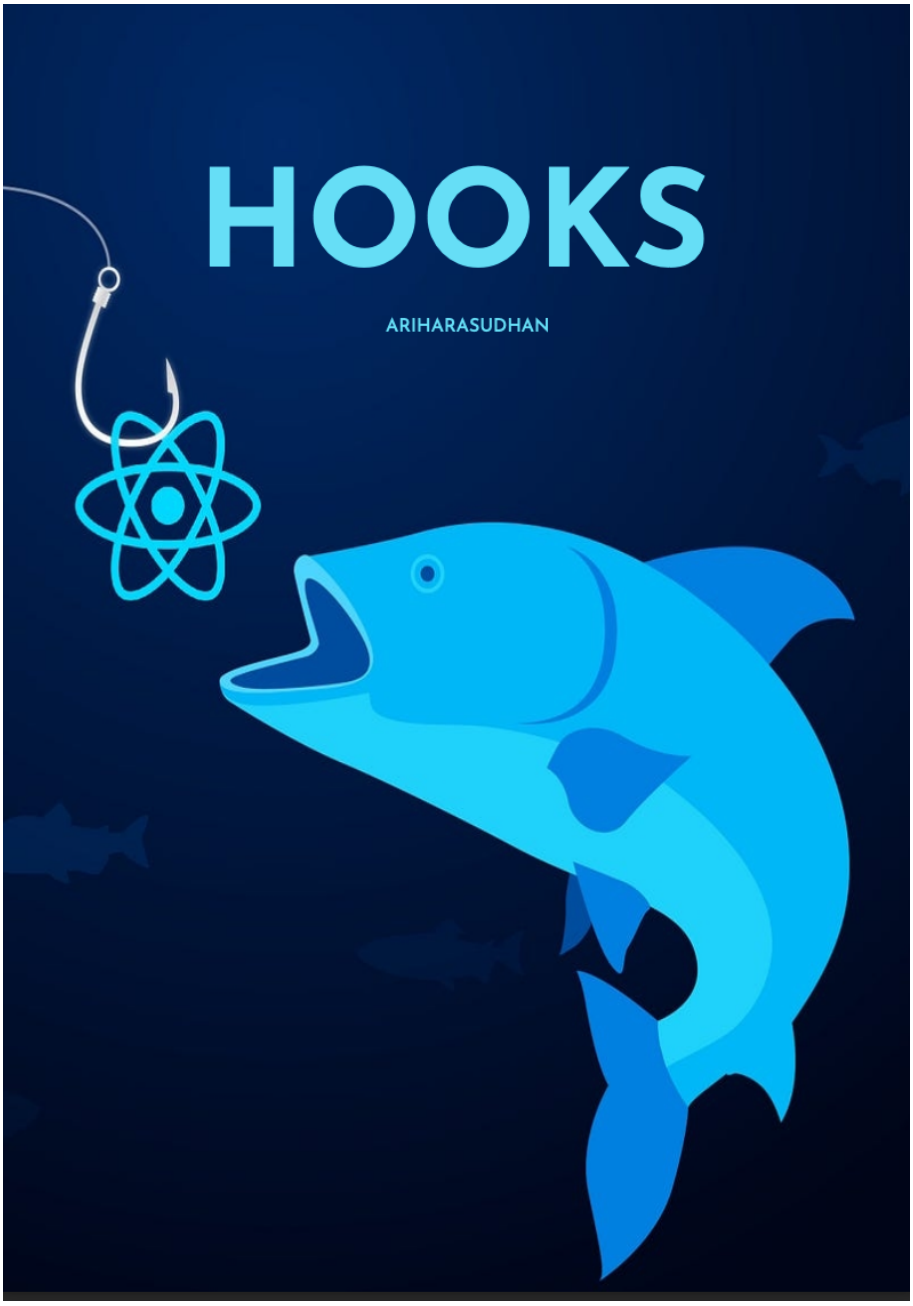
On clicking the Log in,

Welcome back, User!

Log out

☆ HOOKS

The story gets interesting with The Concept of Hooks. It is available in:



Kindly visit https://arihara-sudhan.github.io/books and download it. Hooks concept is explained completely in this book.

☆ ROUTING

Routing is handled primarily using the React Router library. React Router enables to define routes in application, handle navigation between different views, and manage parameters in the URL.

Install it using the following command:

> npm install react-router-dom

```jsx
import React from 'react';
import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';
import Home from './components/Home';
import About from './components/About';
import Contact from './components/Contact';

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </Router>
  );
}
```

<Router>: The top-level wrapper that enables routing in the application.

<Routes>: A container for all the route definitions.

<Route>: Defines a specific route.

The path attribute specifies the URL path. The element attribute specifies the React component to render for the given path.

So, here, when we go to /, Home Component is rendered. When we go to /about, About Component is rendered. When we got to /contact, Contact Component is rendered.

Now, let's see how we can perform navigation.

```jsx
import React from 'react';
import { Link } from 'react-router-dom';

function Navbar() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
      <Link to="/contact">Contact</Link>
    </nav>
  );
}
```

This prevents a full page reload, maintaining the single-page application behavior.

We can also create Dynamic Routes. For example, look at the following:

```
function UserProfile({ params }) {
  return <h1>Welcome, User {params.id}</h1>;
}

<Route path="/user/:id" element={<UserProfile />} />
```

This parameter can be retrieved using useParams hook.

```
1   import { useParams } from 'react-router-dom';
2
3   function UserProfile() {
4     const { id } = useParams();
5     return <h1>Welcome, User {id}</h1>;
6   }
```

To perform programmatic navigation, Use the useNavigate hook.

```
import { useNavigate } from 'react-router-dom';

function Login() {
  const navigate = useNavigate();
  function handleLogin() {
    navigate('/dashboard');
  }
  return <button onClick={handleLogin}>Login</button>;
}
```

We can also perform Nested Routing.

For components with sub-routes:

```
<Route path="/dashboard/*" element={<Dashboard />} />
```

That component would look like,

```
function Dashboard() {
  return (
    <div>
      <h1>Dashboard</h1>
      <Routes>
        <Route path="analytics" element={<Analytics />} />
        <Route path="settings" element={<Settings />} />
      </Routes>
    </div>
  );
}
```

THANK YOU