A component based library.

Use of createRoot to add a react component in your html. So only that component will be rendered using react.

Components:

A piece of the UI that has it’s own logic and appearance.

A JS func that returns a markup.

```
let content;
if (isLoggedIn) {
  content = <AdminPanel />;
} else {
  content = <LoginForm />;
}
return (
  <div>
    {content}
  </div>
);
```

If you prefer more compact code, you can use the conditional `?` operator. Unlike `if`, it works inside JSX:

```
<div>
  {isLoggedIn ? (
    <AdminPanel />
  ) : (
    <LoginForm />
  )}
</div>
```

When you don’t need the `else` branch, you can also use a shorter logical `&&` syntax:

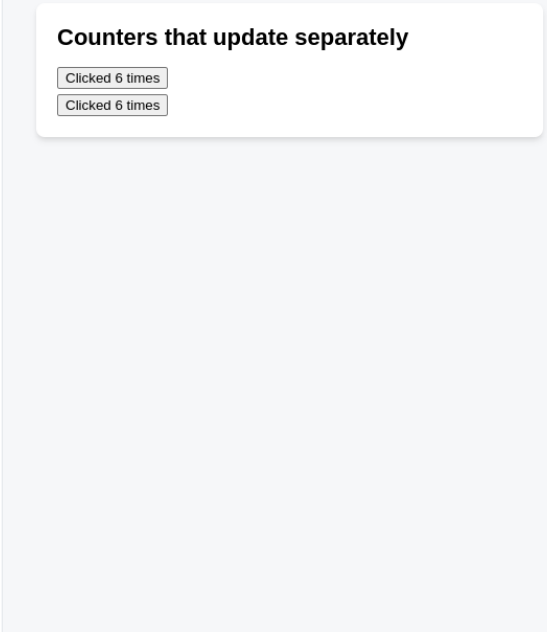
```
<div>
  {isLoggedIn && <AdminPanel />}
</div>
```

React will call your component function again. This time, `count` will be `1`. Then it will be `2`. And so on.

If you render the same component multiple times, each will get its own state. Click each button separately:

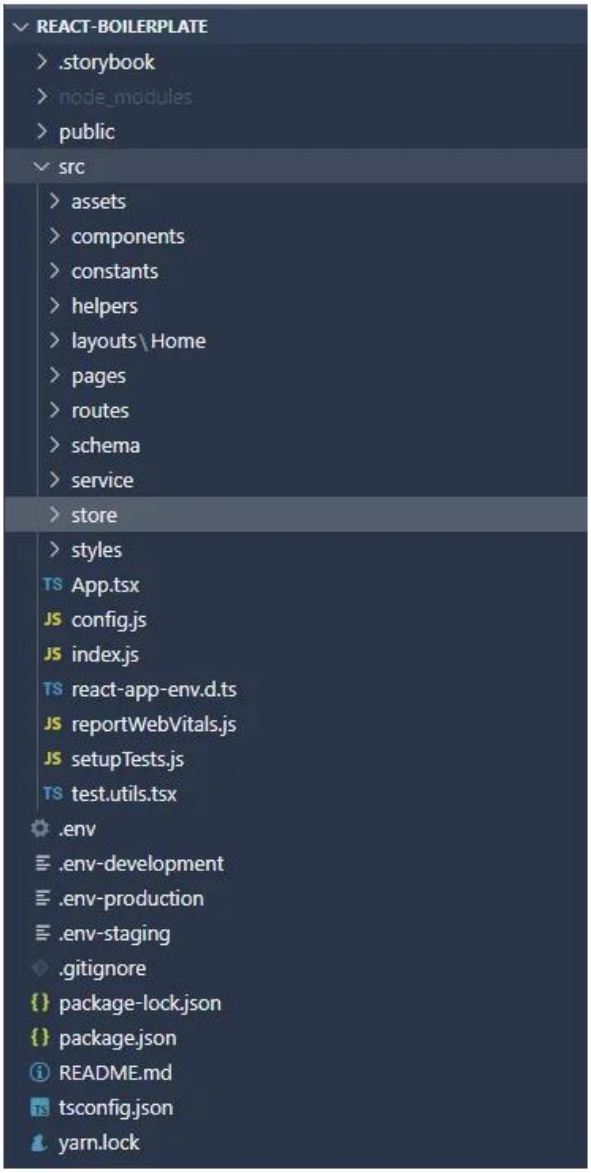
App.js — Download Reset Fork

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

export default function MyApp() {
  return (
    <div>
      <h1>Counters that update separately</h1>
      <MyButton />
      <MyButton />
    </div>
  );
}

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

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      Clicked {count} times
    </button>
  );
}
```



Functions starting with ‘use’ are called hooks

Difference between Library and Framework		
Basis	LIBRARY	FRAMEWORK
Meaning	Library is collection of reusable functions used by computer program	Framework is a piece of code that dictates the arcitecture of project
Inversion of control	With a library , you are in-charge, means you can choose where and when you want to insert or use the library.	In a framework, the framework is in-charge, not you, means a framework tells you where to put a specific part of your code
Function	They are important in program linking and binding process	In a framework , provided standard way to build and deploy applications
Flexiblity	Libraries are more flexible with greater degree of control	Frameworks are enforced structure and standards.
Example	React.js, Jquery is a javascript library	Angular js, Vue js is javascript framework



Components in React:

1. Functional: takes props as input and returns JSX elems
2. Class: Rarely used.

What is the Virtual DOM?

he Virtual DOM is an abstraction of the Real DOM, created and maintained by JavaScript libraries such as React. The Virtual DOM is a lightweight copy of the Real DOM, which allows for faster updates and improved performance. When a user interacts with a web page, React updates the Virtual DOM, compares it with the previous version, and only updates the Real DOM with the necessary changes. This process is known as Reconciliation.

React maintains two Virtual DOM at each time, one contains the updated Virtual DOM and one which is just the pre-update version of this updated Virtual DOM. Now it compares the pre-update version with the updated Virtual DOM and figures out what exactly has changed in the DOM like which components have been changed. This process of comparing the current Virtual DOM tree with the previous one is known as **'diffing'**. Once React finds out what exactly has changed then it updates those objects only, on real DOM.

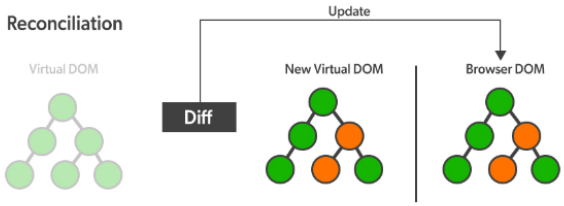
Virtual DOM Key Concepts :

- Virtual DOM is the virtual representation of Real DOM
- React update the state changes in Virtual DOM first and then it syncs with Real DOM
- Virtual DOM is just like a blueprint of a machine, can do changes in the blueprint but those changes will not directly apply to the machine.
- Virtual DOM is a programming concept where a virtual representation of a UI is kept in memory synced with "Real DOM " by a library such as ReactDOM and this process is called reconciliation
- Virtual DOM makes the performance faster, not because the processing itself is done in less time. The reason is the amount of changed information – rather than wasting time on updating the entire page, you can dissect it into small elements and interactions

What is Diffing Algorithm ?

Last Updated : 13 Oct, 2023

Diffing Algorithm in React JS differentiates the updated and previous DOM of the application. DOM stores the components of a website in a tree structure. React uses virtual DOM which is a lightweight version of the DOM. The only difference is the ability to write the screen like the real DOM, in fact, a new virtual DOM is created after every re-render.



What is Diffing Algorithm in React ?

Diffing short for Differences Algorithm is used to differentiate the DOM Tree for efficient updates. React utilize diffing algorithm to identify the changes in the newly created virtual dom and previous version of dom after any changes are made.

How Diffing Algorithm Works?

`constructor(props)`

The constructor runs before your class component *mounts* (gets added to the screen). Typically, a constructor is only used for two purposes in React. It lets you declare state and bind your class methods to the class instance:

Mounting

Mounting means putting elements into the DOM.

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

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

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

Updating

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

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

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

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

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

Unmounting

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

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

- `componentWillUnmount()`

Certainly! Here are the specific uses of each of these lifecycle methods in React:

1. **componentDidMount**:
   - Use Case: Initialization, fetching data, setting up subscriptions.
   - When It's Called: After the component has been rendered to the DOM.
   - Purpose: To perform any necessary setup, such as initializing state, fetching data from APIs, or setting up event listeners.
   - Example: Fetching data from a remote server and updating the component's state with the fetched data.
2. **componentDidUpdate**:
   - Use Case: Responding to prop or state changes, updating the DOM in response to component updates.
   - When It's Called: After a component's props or state have changed and the component has been re-rendered.
   - Purpose: To perform side effects in response to prop or state changes, such as making network requests based on new props or updating the DOM based on state changes.
   - Example: Updating a list of items displayed in the component when the data prop changes.
3. **componentWillUnmount**:
   - Use Case: Cleanup, removing event listeners, canceling network requests, releasing resources.
   - When It's Called: Before a component is removed from the DOM.
   - Purpose: To perform any necessary cleanup tasks before the component is unmounted, such as removing event listeners or canceling network requests.
   - Example: Removing event listeners added in componentDidMount to prevent memory leaks.

These lifecycle methods are essential for managing side effects, updating the UI, and cleaning up resources in React components.

1. **componentDidMount**:
   - Arguments: None.
   - Usage: No arguments are passed to componentDidMount. It's typically used for setting up initial state, fetching data, or initializing subscriptions.
2. **componentDidUpdate**:
   - Arguments:
     - prevProps: Represents the previous props object before the most recent update.
     - prevState: Represents the previous state object before the most recent update.
     - snapshot (optional): Represents a value returned by the getSnapshotBeforeUpdate method, if implemented.
   - Usage: You can compare the current props and state with the previous ones to perform conditional side effects or updates.
3. **componentWillUnmount**:
   - Arguments: None.
   - Usage: No arguments are passed to componentWillUnmount. It's typically used for cleanup tasks such as removing event listeners or canceling network requests before the component is unmounted.

ChatGPT

In React class components, the `**state**` is a JavaScript object that represents the internal data of the component. It is used to store and manage data that can change over time and affect the rendering of the component. When the state of a component changes, React automatically re-renders the component to reflect the updated state.

The `**state**` object is initialized in the constructor of the class component using `**this.state = { /* initial state */ }**`, and it can be updated using the `**setState()**` method provided by React. The `**setState()**` method merges the new state with the existing state, and triggers a re-render of the component with the updated state.

Here's a simple example of using state in a React class component:

ChatGPT

These fields in the `package-lock.json` file serve the following purposes:

1. **Version**: Ensures that a specific version of a package is installed, allowing developers to maintain consistency and predictability in their projects. It helps prevent unintended upgrades or downgrades of dependencies, reducing the risk of compatibility issues.
2. **Resolved**: Provides transparency into the exact source from which a package was fetched during installation. This helps ensure the reliability and security of the package by allowing developers to verify its origin and authenticity.
3. **Integrity**: Guarantees the integrity and authenticity of the installed package by verifying that its content matches the expected cryptographic hash. This prevents tampering or corruption of the package during transmission or installation, enhancing the overall security of the dependency management process.

ChatGPT

When a new project is downloaded, the dependencies are typically installed using both `package.json` and `package-lock.json`. Here's how it usually works:

1. **package.json**: This file contains the list of dependencies required for the project along with their version ranges. When you run `npm install` or `yarn install`, the package manager reads this file and installs the dependencies listed in it.
2. **package-lock.json**: This file is used by npm (or yarn) to lock down the specific versions of dependencies installed in your project. It ensures that the same versions of dependencies are installed across different machines or environments. When you run `npm install` or `yarn install`, the package manager looks at this file to determine the exact versions of dependencies to install.

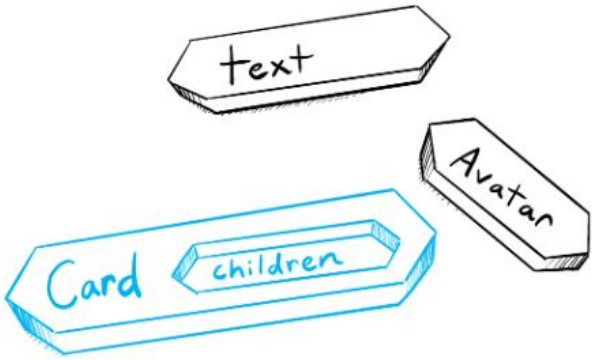
So, both files are important for managing dependencies in a project. While `package.json` specifies which dependencies are needed, `package-lock.json` ensures that the same versions of dependencies are installed consistently across different environments.

Props:

Used to pass args or data to components. Passed in a way similar to that of HTML-tag attr. They are read only and cannot be changed.

You can think of a component with a `children` prop as having a "hole" that can be "filled in" by its parent components with arbitrary JSX. You will often use the `children` prop for visual wrappers: panels, grids, etc.

Illustrated by Rachel Lee Nabors



ChatGPT

React will rerender a component under the following circumstances:

1. **Props Change**: If the component receives new props from its parent component, it will rerender to reflect the changes in the props.
2. **State Change**: When the component's state changes, React will trigger a rerender to update the component's UI based on the new state.
3. **Force Update**: You can explicitly force a component to rerender by calling `` `this.forceUpdate()` `` within the component. However, it's generally recommended to avoid using `` `forceUpdate()` `` and instead rely on props and state changes to trigger rerenders.
4. **Context Changes**: If the component subscribes to changes in a context using the `` `useContext()` `` hook (or its equivalent in class components), it will rerender when the context value changes.
5. **Parent Component Rerender**: If a component's parent component rerenders for any reason, all of its children will also rerender, including the component in question.
6. **Lifecycle Methods**: Certain lifecycle methods like `` `shouldComponentUpdate()` `` can be implemented to control when a component should rerender based on specific conditions.

React uses a virtual DOM and a reconciliation process to efficiently update the actual DOM only when necessary, minimizing unnecessary rerenders for performance optimization.

React will also try to group or batch setState calls into a single call, which leads us to our first “gotcha”:

```
// assuming this.state = { value: 0 };
this.setState({ value: this.state.value + 1});
this.setState({ value: this.state.value + 1});
this.setState({ value: this.state.value + 1});
```

After all the above calls are processed `this.state.value` will be 1, not 3 like we would expect! To get around that ...

setState accepts a function as its parameter

If you pass a function as the first argument of setState, React will call it with the at-call-time-current state and expect you to return an `Object` to merge into state. So updating our example above to:

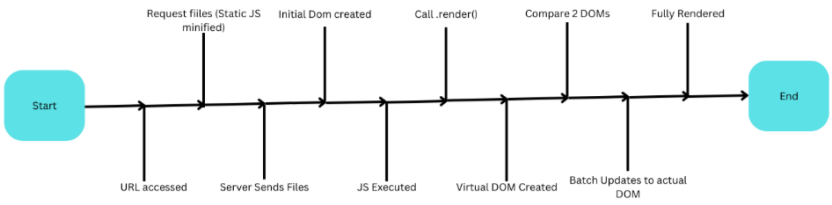
```
// assuming this.state = { value: 0 };
this.setState((state) => ({ value: state.value + 1}));
this.setState((state) => ({ value: state.value + 1}));
this.setState((state) => ({ value: state.value + 1}));
```

Will give us `this.state.value = 3` like we expected in the first place. Remember to always use this syntax when updating state to a value, which is computed based on previous state!

Top highlight

doesn't return a promise.

> *Async functions always return a promise. If the return value of an async function is not explicitly a promise, it will be implicitly wrapped in a promise.*



Functional Components:

- **Simplicity:** Functional components are simpler and easier to understand than class components. They don't have the added complexity of lifecycle methods, state, and this binding.
- **Performance:** Functional components are more performant than class components. They don't have the added overhead of creating a new instance for each render and also functional components can use the React Hooks which makes it more performant.
- **Easier to test:** Functional components are easier to test than class components. Because they are just plain JavaScript functions, they can be tested with standard JavaScript testing tools like Jest.
- **Easier to reuse:** Functional components are easier to reuse than class components. Because they are just plain JavaScript functions, they can be easily reused across different parts of your application.
- **Easier to reason about** Functional components are easier to reason about than class components. Because they are just plain JavaScript functions, their behavior is more predictable and easier to understand.
- **Hooks:** Functional components can use the React hooks, which allows you to use state and other React features in functional components, while class components can't. Refer to this GFG Article to have better a understanding of Hooks – Link.

UseMemo:

```jsx
function ExpensiveCalculation({ a, b }) {
  // Define a function to perform an expensive calculation
  const computeExpensiveValue = (a, b) => {
    console.log('Performing expensive calculation...');
    return a * b;
  };

  // Use useMemo to memoize the result of the expensive calculation
  const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

  return (
    <div>
      <p>Value of a: {a}</p>
      <p>Value of b: {b}</p>
      <p>Result of expensive calculation: {memoizedValue}</p>
    </div>
  );
}

function App() {
  const [a, setA] = useState(5);
  const [b, setB] = useState(10);

  const handleChangeA = (e) => {
    setA(parseInt(e.target.value));
  };

  const handleChangeB = (e) => {
    setB(parseInt(e.target.value));
  };

  return (
    <div>
      <input type="number" value={a} onChange={handleChangeA} />
      <input type="number" value={b} onChange={handleChangeB} />
      <ExpensiveCalculation a={a} b={b} />
    </div>
  );
```

ChatGPT

No, `useMemo` itself is not asynchronous. It executes synchronously during the render phase of React's reconciliation process.

If the function passed to `useMemo` is synchronous and takes some time to execute, it will block the main thread during that time, potentially leading to performance issues such as a sluggish user interface or unresponsive behavior.

It's important to note that `useMemo` is not intended for handling asynchronous operations. If the expensive calculation involves asynchronous operations (such as fetching data from an API or performing I/O operations), you should use asynchronous methods such as `async/await`, `Promise`, or `useEffect` (if you need to perform the operation side-effect), depending on the specific use case.

Here's an example of how you might handle an asynchronous operation using `useEffect` and state:

jsx Copy code

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

function ExpensiveCalculation({ a, b }) {
```

UseReducer:

```jsx
import React, { useReducer } from 'react';

// Reducer function
const reducer = (state, action) => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
};

function Counter() {
  // Initial state
  const initialState = { count: 0 };

  // useReducer hook
  const [state, dispatch] = useReducer(reducer, initialState);

  // Action functions
  const increment = () => {
    dispatch({ type: 'increment' });
  };

  const decrement = () => {
    dispatch({ type: 'decrement' });
  };

  return (
    <div>
      <p>Count: {state.count}</p>
```

Message ChatGPT...

ChatGPT

`useReducer` is a React Hook that is used for managing state in a more complex way compared to `useState`. While `useState` is typically used for managing independent pieces of state, `useReducer` is often used for managing state that involves multiple actions or transitions.

Here's how `useReducer` works:

1. **Reducer Function**: The `useReducer` hook accepts a reducer function as its first argument. The reducer function takes two arguments: the current state and an action, and returns the new state based on the action. The signature of the reducer function is `(state, action) => newState`.
2. **Initial State**: You also need to provide an initial state value as the second argument to `useReducer`.
3. **Dispatch Function**: `useReducer` returns an array with two elements: the current state and a dispatch function. The dispatch function is used to dispatch actions to the reducer, which in turn updates the state based on those actions.

Here's an example of how you can use `useReducer` to manage state for a simple counter:

1. **useState**:
   - **Simple State**: Use `useState` for managing simple, independent state variables where the state transitions are straightforward.
   - **Single Value**: Use `useState` when you only need to manage a single piece of state.
   - **Local Component State**: `useState` is typically used for managing state that is specific to a particular component and doesn't need to be shared with other components.

Example use cases for `useState`:

- Managing a boolean flag (e.g., visibility, toggles).
- Tracking input field values.
- Storing a counter value.

2. **useReducer**:
   - **Complex State Logic**: Use `useReducer` for managing more complex state logic that involves multiple actions or transitions.
   - **Multiple Related Values**: Use `useReducer` when you need to manage multiple related state variables together in a single state object.
   - **Shared State**: `useReducer` can be useful for managing global or shared state that needs to be accessed by multiple components.

Example use cases for `useReducer`:

- Managing state for forms with validation and submission.
- Implementing complex UI components with different states and interactions.
- Handling state for a shopping cart with multiple actions (add item, remove item, update quantity).

ChatGPT

`useRef` is a React Hook that provides a way to create mutable references to elements or values that persist across renders of a functional component. It allows you to access the DOM nodes directly or to store mutable values that persist between renders without causing re-renders.

Here's how `useRef` works:

1. **Creating a Ref**: You create a ref by calling the `useRef` function. The `useRef` function returns a mutable object with a `.current` property that can be assigned a value.
2. **Accessing the Current Value**: The `.current` property of the ref object holds the current value. You can read and modify the value by accessing `.current`.
3. **Persistence**: The value of a ref persists across re-renders of the component. Changing the value of a ref does not trigger a re-render.
4. **DOM References**: `useRef` is commonly used to get references to DOM elements. You can assign the ref object to the `ref` attribute of a JSX element to gain access to the corresponding DOM node.

Here's an example of how you can use `useRef` to get a reference to a DOM element:

The first rule was particularly interesting to me. It seemed easy enough to follow, but why did the rule exist? What exactly happens if Hooks aren't called in the exact same order every single time a component is rendered? How convenient that the doc writers linked to an explainer! To summarize the explanation, if Hooks aren't called in the same order each time a component is rendered, you could have cascading failures where you miss one Hook and none of the subsequent ones are recognized properly. The Hook calls could end up shifted and you'll get unexpected bugs.

Well, that seems pretty bad. Let's definitely not allow that to happen. So we now know why the rule shouldn't be broken, but not why things break. I still really wanted to understand why call order was so important to Hooks. To figure that out I decided to jump into the source code. I started here where the Hooks are defined and eventually followed it over to here where the Dispatcher is. That's where I saw this note:

```
Hooks are stored as a linked list on the fiber's memoizedState field.
The current hook list is the list that belongs to the current fiber.
The work-in-progress hook list is a new list that will be added to
the work-in-progress fiber.
```