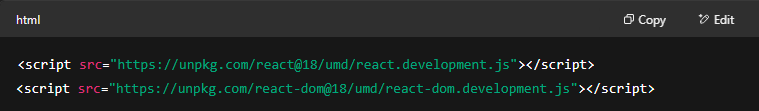
19/5/2025

**CDN** stands for **Content Delivery Network**. A **CDN link** refers to a URL that delivers files like JavaScript libraries, CSS stylesheets, fonts, etc., directly from a **distributed network of servers** instead of your local system.

Using a CDN link means you're loading a file (e.g., React library) **from the internet** rather than **bundling it with your project files**.



These lines load **React and ReactDOM** from [**unpkg**](https://unpkg.com/), which is a CDN that hosts npm packages.

When you use React via CDN:

1. **No installation needed** – You don't need to run npm install react or use tools like Webpack, Vite, or Parcel.
2. **The React library loads at runtime** from the URL, directly into the browser.
3. **React is available globally** in the browser via window.React and window.ReactDOM.
4. You typically write code in a single HTML file, often using inline or linked JS.

When you load React from a CDN:

* Your browser fetches the script from the CDN server.
* It caches the script, so subsequent loads are faster (even across different websites using the same version).
* React is added to the global scope, so you can start using it directly.

CDNs are ideal for:

* Learning or rapid prototyping
* Embedding React into non-React apps
* Avoiding build steps for very lightweight tools

In the above CDNS, the first link is the actual react package and react link.

**🔹 \_\_SECRET\_INTERNALS\_DO\_NOT\_USE\_OR\_YOU\_WILL\_BE\_FIRED — Summary**

* **Private React internals** used by React itself and DevTools.
* **Not meant for developers** — unstable and can break anytime.
* Exists to support **internal logic** like rendering, scheduling, and hooks.
* **Never use in your app** — always stick to public React APIs.

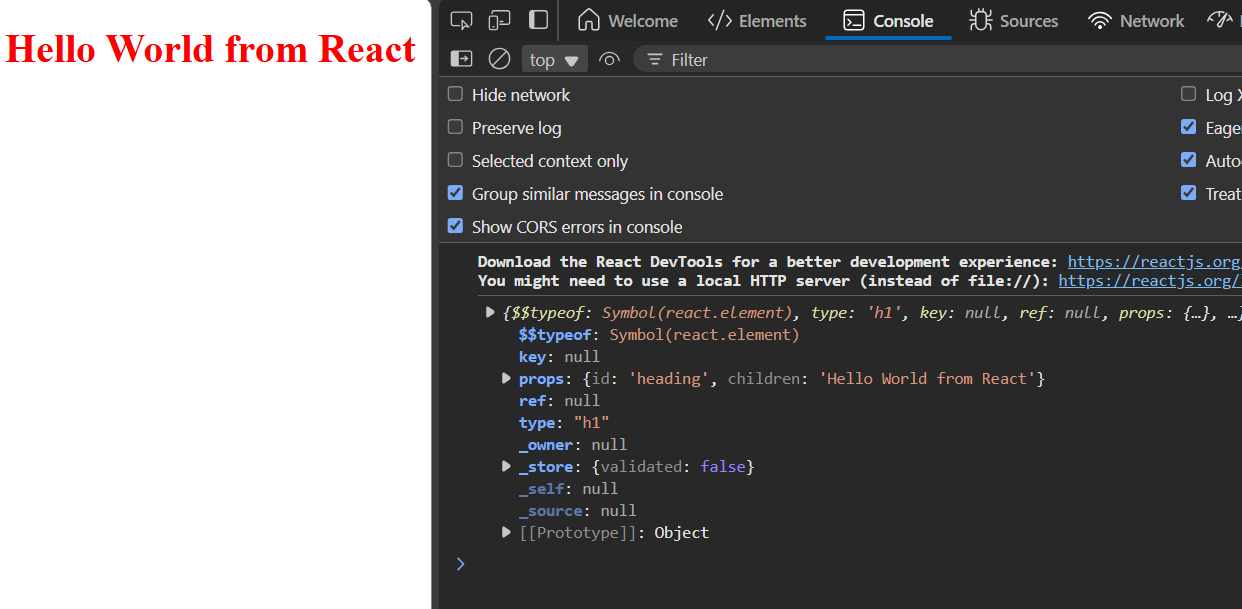
✅ Good for understanding React internals (for study).  
❌ Not for use in production or regular projects.

This is similar to reaching inside an engine while it's running — **you might get things done**, but you're very likely to mess something up.

CDNs use your IP to estimate location and route your request to the **nearest or fastest edge server** using **Geo-DNS** and **Anycast**, reducing load times and latency.

21/5/2025

What does the React.createElement() actually return to us?



In react-

In the context of React elements, type, typeof, and $$typeof serve distinct purposes related to identifying and validating the element.

* **type**:

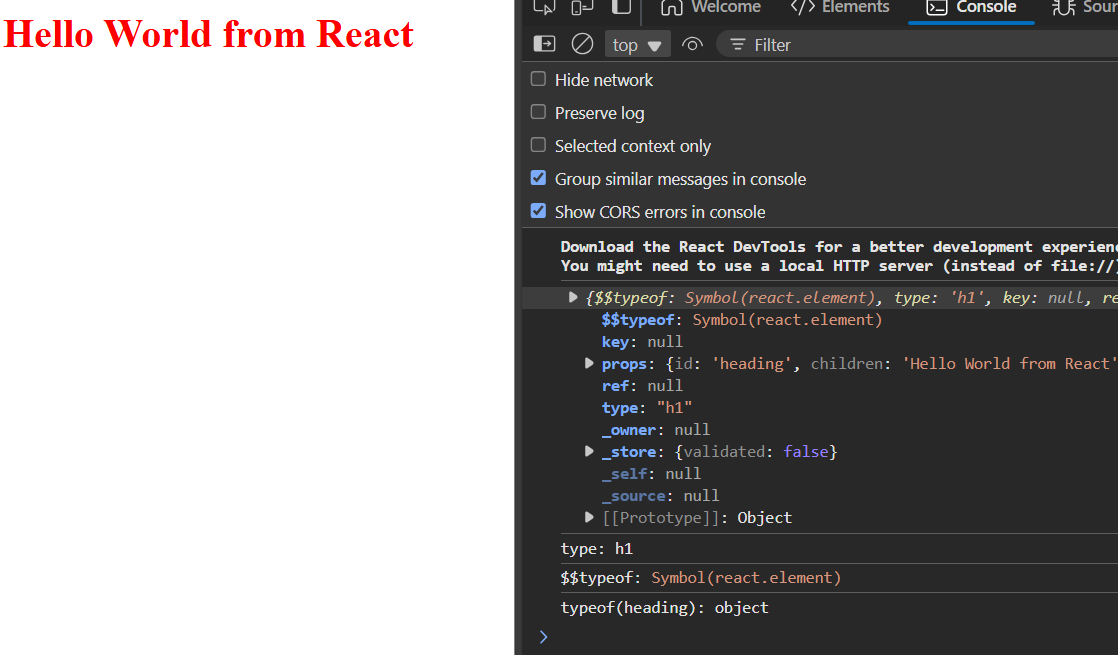
It specifies the kind of React element, which can be either a string representing an HTML tag name (e.g., 'div', 'span') or a React component (function or class).

* **typeof**:

It is a JavaScript operator that returns a string indicating the data type of a value. When applied to a React element, it generally returns 'object' because React elements are plain JavaScript objects.

* **$$typeof**:

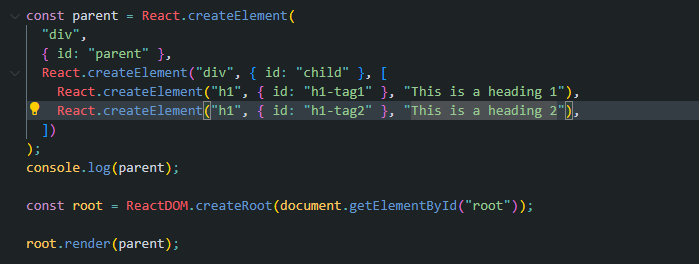
It is a special property added by React to elements to uniquely identify them as React elements. It uses a Symbol if available, or a unique number as a fallback. This property helps prevent cross-site scripting (XSS) attacks by ensuring that only valid React elements are processed.



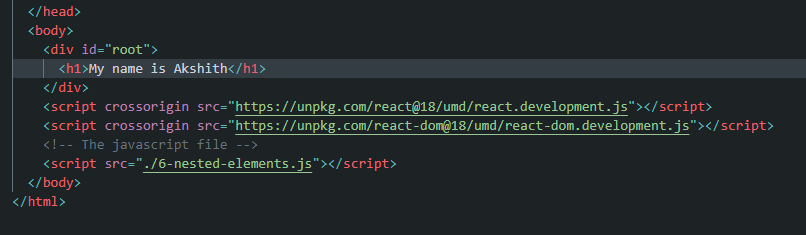
Creating nested elements in JS using react.createElement



For passing multiple children create an array of children and pass it as the third argument to the parent React element.



React replaces whatever we write inside the root element when we use the render function. For example the h1 with my name will be replaced by the parent element when the render function is executed.



Order of execution of the scripts also matter. If I have my js script before react is imported then I will get React not found error.

Q. What is crossorigin attribute in the script tag of html

When you add a <script> tag in HTML to load JavaScript, it often looks like this:

<script src="https://example.com/script.js"></script>

This tells the browser: "Hey, go to example.com and download this JavaScript file."

But if the file is **from another website (a different origin)**, the browser is a bit more careful—because it wants to keep you safe from security risks. That's where the crossorigin attribute comes in.

**So, what does crossorigin do?**

It tells the browser **how to behave when getting files from another website**.

There are two common options:

| **Value** | **What it means in simple terms** |
| --- | --- |
| anonymous | Don't send any info about the user (like cookies) with the request. |
| use-credentials | Do send info like cookies or login tokens with the request. |

**Why would you need it?**

Mainly for **security**—especially if you're using something called **Subresource Integrity (SRI)**. SRI lets you check that the file hasn't been tampered with. But for it to work with files from another site, you must use crossorigin="anonymous".

When you're using elements like <img>, <script>, <video>, etc., they often **load files (resources)** like images, videos, stylesheets, or JavaScript from somewhere else — sometimes from a **different domain** (cross-origin).

The **crossorigin attribute** helps you tell the **browser**:

"Hey, here's how you should behave when loading this resource from a different site."

This is important because of **CORS (Cross-Origin Resource Sharing)** — a security feature in browsers that controls **what kinds of cross-origin requests are allowed.**

**What is CORS ?**

**CORS** is a **security feature** built into **web browsers**. It controls **which websites** are allowed to **access resources** (like APIs, fonts, or files) from a **different domain**.

To conclude:

If your browser has **CORS enabled** (which all modern browsers do by default):

* When a webpage tries to request resources from a **different origin** (domain/protocol/port),
* The browser **checks** the server’s response headers for **CORS permissions** (like Access-Control-Allow-Origin),
* If the server **allows** your site’s origin in these headers, the browser **lets the webpage access the resource**,
* If the server **does not allow** it (header missing or origin not allowed), the browser **blocks the response**, preventing access to the data for security reasons.

This protects you from malicious sites trying to steal data from other sites without permission.

**Think of it like this:**

* Your browser is like a **security guard** who won’t let any strangers take data from a building (the server).
* The **server is the building owner** who gives the guard a list of **approved visitors** (allowed origins).
* Without the owner’s permission (CORS headers), the guard blocks access—even if the visitor asks nicely.

Finally to conclude –

When you use crossorigin="anonymous":

* Your browser makes the cross-origin request **without sending credentials** like cookies, HTTP authentication, or client-side SSL certificates.
* This means your **personal info or session data isn’t shared** with that external server, even if it *allows* cross-origin requests via CORS headers.
* It’s a way to **limit what data the server can see about you**, protecting your privacy when fetching resources from third-party or untrusted origins.

So, even if the server has CORS enabled and says “anyone can access my resource,” your credentials stay private unless you explicitly use a different crossorigin mode (like "use-credentials").

**Declarative**

React is **declarative** because instead of telling the browser *how* to update the UI step-by-step, you simply describe **what the UI should look like** based on the current state — and React handles the "how" for you.

* You describe **what** the UI should look like.
* React figures out **how** to efficiently update the DOM.
* This makes your code more predictable and easier to debug.

Vanilla JS is imperative, we instruct the browser how to manipulate the DOM and update it.

22/5/25 – Igniting our application

Trivia -

“ Contrary to popular belief, npm **is not** in fact an acronym for "Node Package Manager"; It is a recursive bacronymic abbreviation for **"npm is not an acronym"** (if the project was named "ninaa", then it would be an acronym). The precursor to npm was actually a bash utility named **"pm"**, which was the shortform name of **"pkgmakeinst"** - a bash function that installed various things on various platforms. If npm were to ever have been considered an acronym, it would be as "node pm" or, potentially "new pm". ”

So this means they had never thought of the full form for npm and it was just npm without any full form or as provided by GitHub you can take it as “node pm”.

(see all notebook notes of npm, npm init, etc)

**📦 package.json: The Project Manifest**

This file is the **main metadata file** for a Node.js project.

**✅ What it contains:**

* Project name, version, author, license
* Scripts (e.g., npm run build, npm test)
* Dependencies and devDependencies (e.g., "react": "^18.2.0")
* Main entry point (main, like index.js)
* Configuration for tools like Babel, ESLint, etc.

**🔧 Purpose:**

* **Defines your app**: What it is, how to run it, and what it needs.
* **Allows others to install dependencies** using npm install.
* **Specifies dependency ranges**, not exact versions (e.g., "^1.2.3" allows patch and minor updates).

**🔒 package-lock.json: The Dependency Snapshot**

This file is **automatically generated by npm** when you install packages.

**✅ What it contains:**

* **Exact versions** of every package installed, including nested dependencies.
* A complete dependency tree.
* Resolved URLs and integrity hashes to ensure consistency.

**🔧 Purpose:**

* **Locks dependencies** to exact versions used in your local environment.
* **Ensures reproducibility** across environments (e.g., teammates or CI/CD pipelines).
* Helps avoid “works on my machine” problems.
* **Improves performance**: npm can skip resolution steps and just follow the lock file.

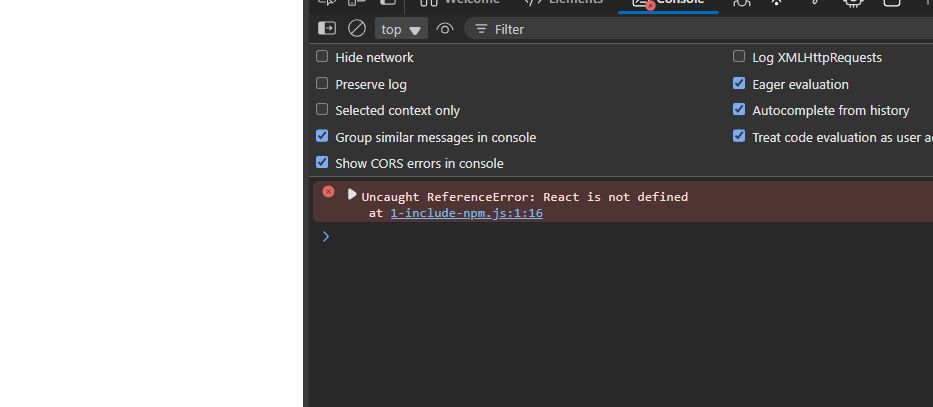
**🔄 Summary Table**

| **Feature** | **package.json** | **package-lock.json** |
| --- | --- | --- |
| Purpose | Project manifest | Dependency version lock |
| Version format | Ranges (^, ~) | Exact versions |
| Editable by developer | Yes | No (auto-generated by npm) |
| Tracks nested deps | No | Yes |
| Committed to git? | ✅ Yes | ✅ Yes |
| Ensures same versions | ❌ Not guaranteed | ✅ Guaranteed |

**🔁 Typical Workflow**

1. You run npm install react.
2. package.json updates with "react": "^18.2.0".
3. package-lock.json records the exact version of React and its sub-dependencies.

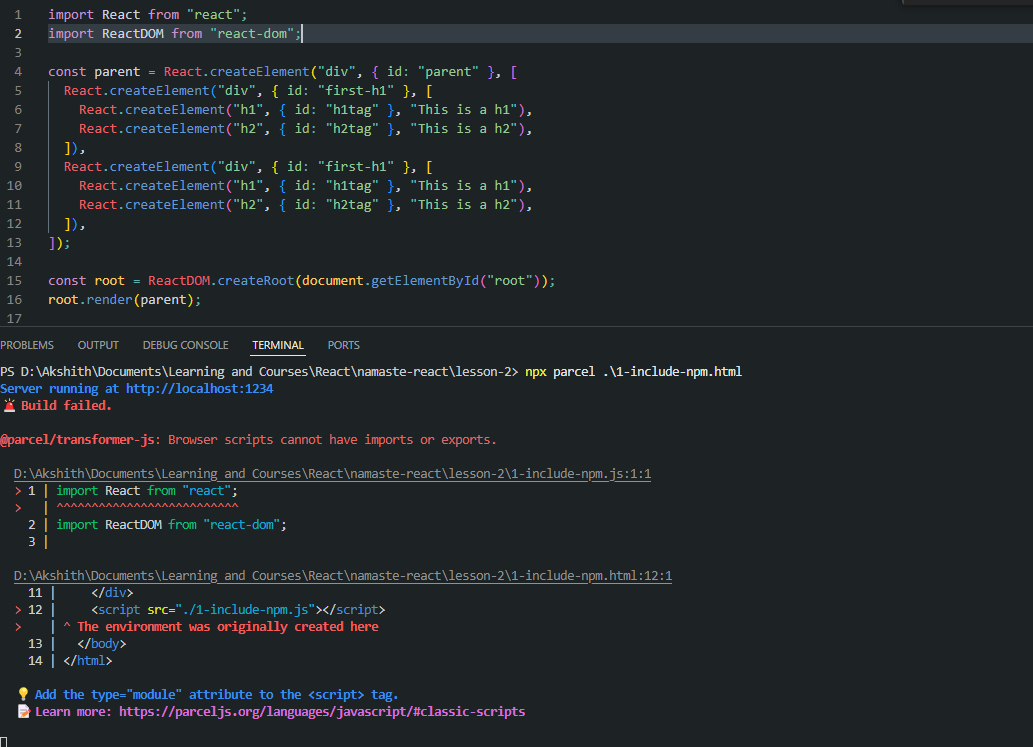
After doing npx parcel index.html, npm I react and then removing the CDN links, if we try seeing our application we get the below error saying, ‘React’ is not defined. This is because we have only installed react but we are not yet using it.



Solution we have to write

Import React from “react” . This “react” is the one installed in node modules.

Upon doing it we get this error –



This is because the application is considering the script in the html as a normal JS and import is not a recognized keyword in the JS browser script.

Normal browser js does not have “import”.

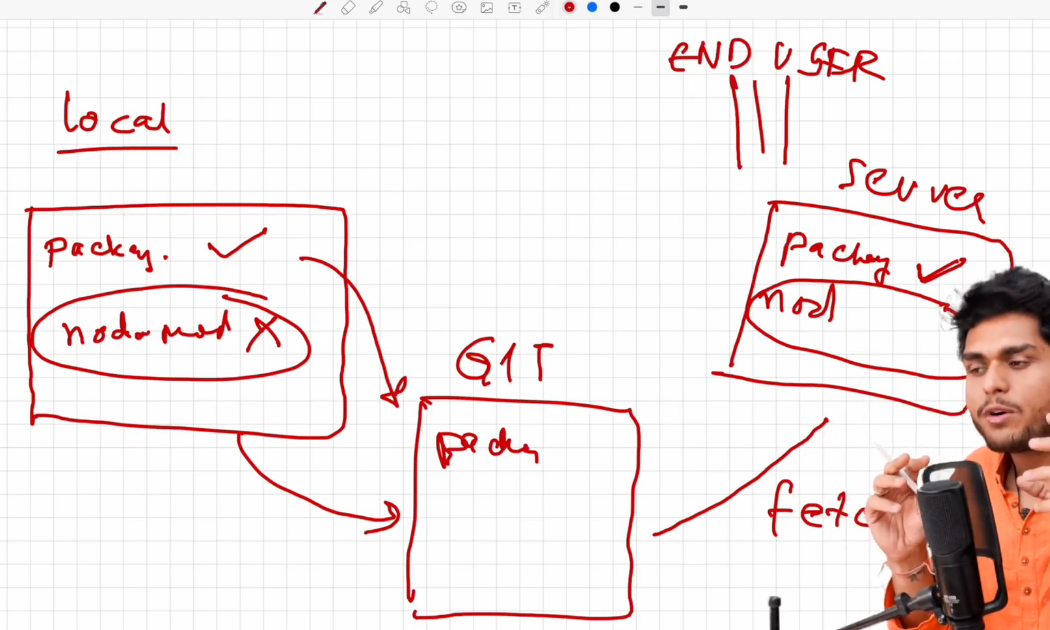
We need to tell the browser that this is not a normal js file but we have to mention type=”module” in our html file In the script tag where we are invoking the react application.

**Now React is coming from ‘react’** which is present in node\_modules, which came from npm.

Power of parcel – (Also read parcel documentation)



Working of Git, local repo and GitHub.



Some features of parcel –

**🔥 What is Hot Module Replacement (HMR)?**

HMR allows you to **update modules in a running app without a full reload**. This is especially useful in development, as it:

* Preserves component state
* Speeds up development workflow
* Reduces context-switching

Parcel (like Vite, Webpack with React Refresh, etc.) handles **Hot Module Replacement (HMR)** and **Fast Refresh** automatically for React apps:

* **Component state is preserved** across edits (like when you're changing JSX or styles).
* The page **doesn't reload** unless the update can't be handled gracefully (e.g., changing hooks order or modifying module-level state).
* No need to manually use module.hot.accept() or module.hot.dispose() unless you're doing **non-React custom logic** (like global timers, WebSocket connections, or canvas animations outside the React tree).

**🔁 How HMR Works**

* Bundlers (like Parcel/Vite) watch for file changes, rebuild **only the changed module and its dependencies**, and use **WebSocket** to send updates to the browser, which replaces them in-place without a full reload.

**🧠 How State Is Preserved**

* Frameworks like React store component state in memory (via hooks, context, etc.), so as long as the component **isn’t unmounted**, state is preserved during HMR.

**⚠️ When State Is Lost**

* State can be lost if the component is **unmounted/remounted**, or you change global logic (like Redux store), or you’re not using a framework that supports HMR well.

| **Term** | **Description** |
| --- | --- |
| **Transpiling** | Converts between high-level languages (e.g., JSX → JS) |
| **Compiling** | Converts from high-level to low-level (e.g., C → machine code) |
| **Bundling** | Combines multiple files into one (e.g., with Webpack, Parcel) |
| **Minifying** | Reduces file size (removes whitespace, comments, etc.) |

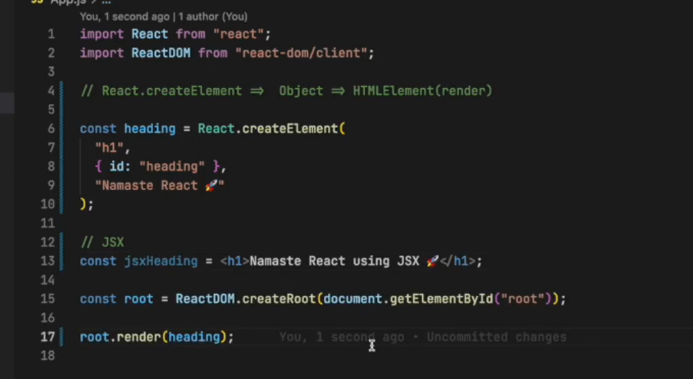
**npx parcel build 1-include-npm.html**  creates an optimized, minified production build of your project. It outputs static files to the dist/ folder, ready for deployment. No dev server runs during this process.

**npx parcel 1-include-npm.html** starts a local development server with hot module replacement (HMR). It serves your files in-memory, updates the browser live as you code, and is not optimized for production.

A **module** is **any JavaScript file that exports something** and can be imported elsewhere. It’s a self-contained unit of code.

A module is a JavaScript file — not specifically a React component, though a React component **can** be a module.

24/5/25

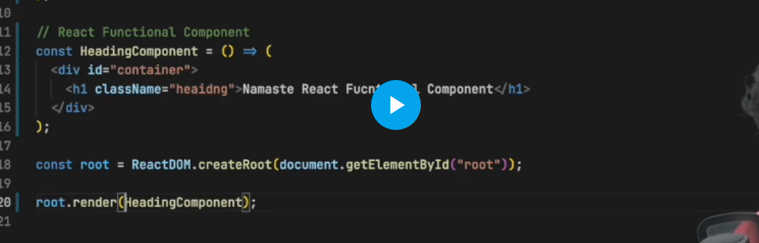


JSX is not html in JS

It is html-like syntax, just looks like html or xml.

25/5/25

We cannot render a react functional component like this as given below



We need to render it like this, within tags- <HeadingComponent/>

(Read notebook and source code)

The full form of JSX is JavaScript XML. It's a syntax extension for JavaScript, primarily used in the React framework, that allows you to write HTML-like syntax within your JavaScript code.

**Cross-site scripting**

Cross-site scripting (XSS) is a web security vulnerability that allows attackers to inject malicious scripts into legitimate websites, which are then executed by other users' browsers. These scripts can steal sensitive information like cookies, hijack sessions, or redirect users to malicious websites.

Here's a more detailed explanation:

* **Injection:**

Attackers inject malicious scripts into a website, often by exploiting vulnerabilities in how the website handles user input.

* **Execution:**

The injected scripts are then executed by a user's browser when they visit the compromised website.

* **Exploitation:**

The malicious scripts can then be used to steal sensitive information, hijack user sessions, or redirect users to malicious sites.

Example:

An attacker might inject a script that redirects users to a fake login page that looks identical to the real one. If a user enters their credentials on the fake page, the attacker can steal their login information.

**How does JSX prevent XSS attacks?**

JSX (JavaScript XML), used primarily in React, **helps prevent Cross-Site Scripting (XSS) attacks** through **automatic escaping** of values embedded in the markup.

When you embed dynamic content in JSX using curly braces ({}), React automatically **escapes the values before rendering** them into the DOM. This means any HTML or JavaScript code injected via a variable is treated as plain text, **not as executable code**.

**XSS happens when an attacker can inject something like that** through user input (e.g., a text box), and it gets interpreted as code.

React **prevents this** by not letting strings like "<img src=x onerror=alert(1)>" become actual HTML — they get escaped.



29/5/2025

[Index as a key is an anti-pattern | Robin Pokorny](https://robinpokorny.com/blog/index-as-a-key-is-an-anti-pattern/)

**Why not use index as a key in React?**

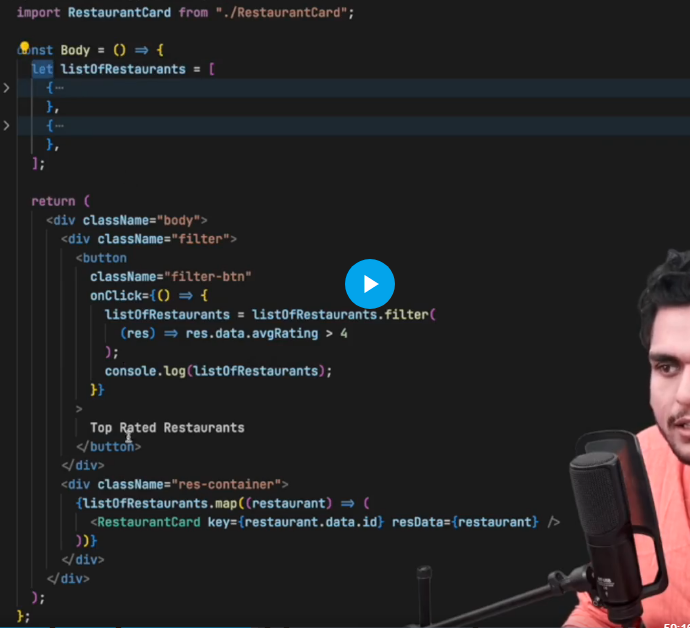
Using the **index** of an item in a list as a key in React is discouraged when the **order of items can change**, or if items can be **added or removed**. This is because React uses the key prop to identify which items have changed, been added, or been removed, enabling it to efficiently update the DOM.

If you use the index as the key:

* When the list order changes, React can't tell which item is which.
* It may reuse the wrong component for a different item.
* This can lead to bugs like incorrect input values, mismatched data, or unexpected UI behavior.

**Always use a stable, unique identifier** (like an id) from your data whenever possible. This ensures that each list item has a persistent identity across renders, even if the order changes.

31/5/2025



IN this case on click of button our data gets updated, but the UI does not get updated. Data is filtered but UI did not update.

Ideally UI should have also changed automatically when data changed

React Hooks

A hook is a JavaScript utility function which is provided by react.

useState() – It is a react hook, which is provided by react to create and manage the state of a react component.

**What is a state variable in React?**

A **state variable** in React is a **piece of data** that is **managed by a component** and can **change over time**. When a state variable is updated, React automatically **re-renders** the component to reflect the changes in the UI. React tracks only state variables.

In functional components, we use the useState hook to create and manage state variables.

If you change a **regular variable** inside a React component (not a state variable), **React will not re-render the UI automatically.** That’s because React doesn’t “watch” normal variables — it only tracks **state** and **props**.

**🧠 What is React Fiber?**

**React Fiber** is the **reconciliation engine** in React — it’s the part of React that is responsible for **rendering components, updating the DOM**, and managing **state and lifecycle methods** efficiently.

It was **introduced in React 16** to rewrite the old React core and improve how updates are handled in the React virtual DOM.

**🚀 Why React Fiber?**

React’s older architecture was **synchronous and recursive**, meaning updates were blocking and couldn’t be interrupted. This caused performance issues for large apps.

**Fiber allows React to:**

* **Pause, resume, or cancel rendering work**
* **Split work into smaller units ("fibers")**
* **Prioritize updates** (e.g., animations over background data)
* Improve **performance and responsiveness**

**🧱 How does it work?**

Fiber breaks rendering into **units of work**, which means:

* React can work on parts of the UI **incrementally**
* It can **pause rendering** if something more urgent comes up (like user input)
* It uses **a tree structure** (Fiber tree) to represent components and their relationships

Each **Fiber node** in the tree contains:

* Component information
* Props and state
* Pointers to child, sibling, and parent
* Effect tags (for knowing what needs to be updated in the DOM)

**⚙️ React Fiber enables:**

* **Concurrent rendering** (React 18+)
* **Suspense** and **lazy loading**
* **Time slicing** for smoother updates
* Better **error handling** (via error boundaries)

**📝 Summary (for assignment):**

React Fiber is the internal engine that powers React’s rendering and reconciliation process. It enables React to break rendering into smaller chunks, prioritize tasks, and make the UI more responsive. Fiber allows features like concurrent rendering, Suspense, and better error handling, making React more efficient and scalable.

**🧪 Scenario Example: Complex List + Input Field**

Imagine this:

You have a React app with:

1. A **search input box**
2. A **very large list of items** that takes time to render (say, 10,000 cards)
3. A **filter** is applied as the user types in the input

**⏱️ Traditional React (Pre-Fiber)**

* When you type into the search input, React needs to re-render the list based on your new search term.
* The entire rendering process happens **synchronously** (in one go).
* If rendering takes 200ms, the browser freezes for 200ms — input lags, and user experience feels janky.

**🚀 React Fiber (Concurrent Mode Enabled)**

React Fiber introduces the ability to **pause, interrupt, and resume** rendering:

**Here's what happens:**

1. **You type a letter** in the search input.
2. React **starts rendering** the filtered list of 10,000 items.
3. You type another letter **before** rendering is complete.
4. React **pauses or even aborts** the current rendering task.
5. It **prioritizes the new user input**, then **resumes or redoes** the rendering with the new input.

So rendering isn’t frozen. React behaves more like an operating system's task scheduler — **doing urgent work first**, delaying the less important work.

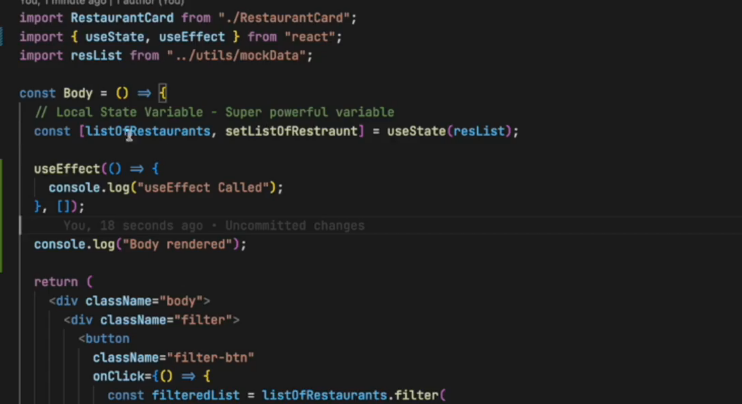
**🧠 How React knows what to pause or resume?**

Fiber tracks the **unit of work** for each component/subtree. So React can:

* Keep track of which part of the component tree it's working on
* Save the state of work-in-progress
* Come back to it later (resume)
* Throw away incomplete work (abort) if newer, higher-priority updates arrive

2/6/25

First the component is rendered and then the useEffect’s callback function is executed.

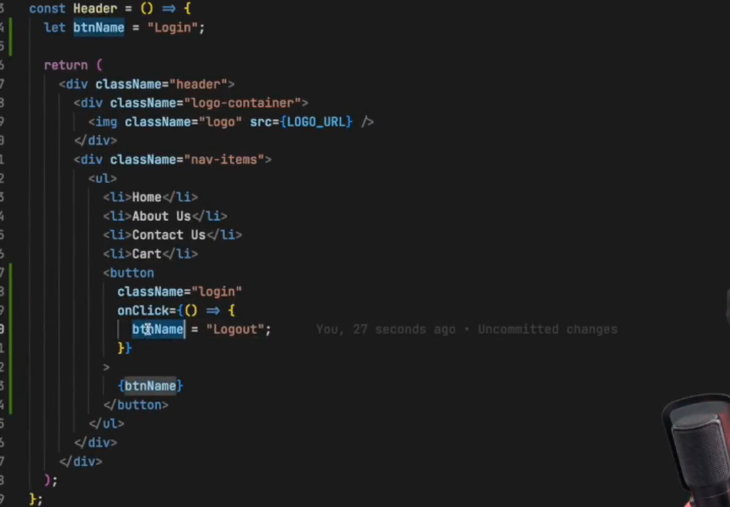


In the above code output –



5/6/2025

Even though here we are updating the local variable ‘btnName’ from Login to Logout, ui will not update as we are not using state variables here.



13/6/2025 – Class based components

Why do we use the super prop and constructor to pass in class based components.

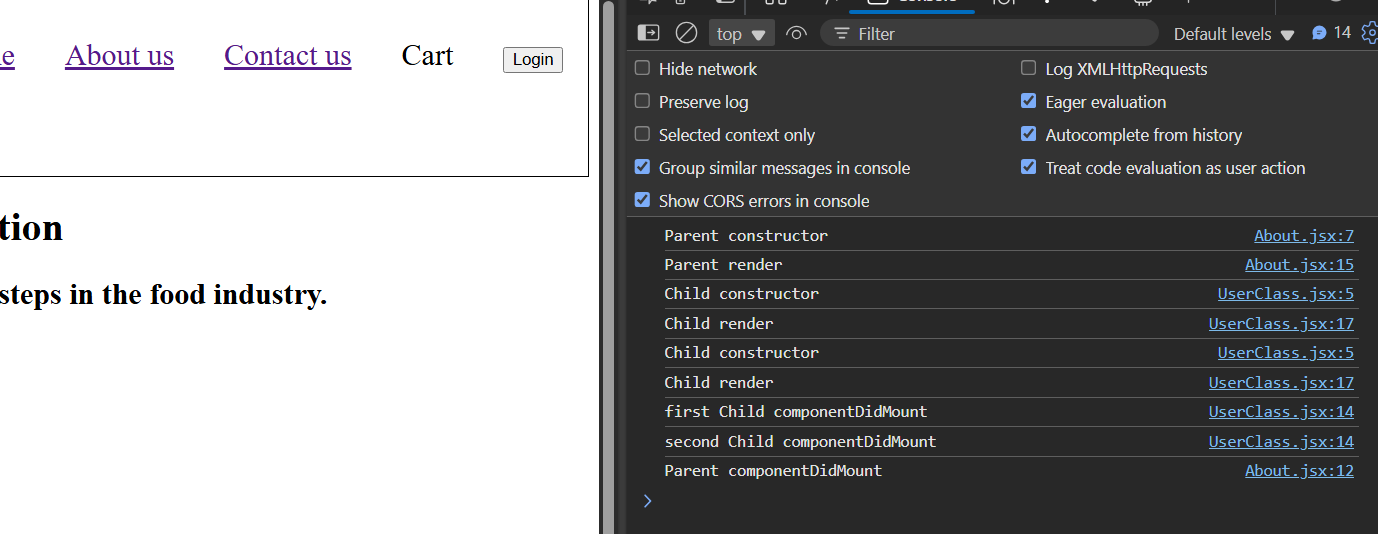
Using the super keyword in the constructor of a React class component allows me to pass the props to the parent class, React.Component. This ensures that the parent class can properly initialize and make use of the props, enabling important functionality like setting up this.props. Without calling super(props), the parent class wouldn’t receive the props, and I wouldn’t be able to access this or this.props inside the constructor, which would lead to errors.

In React class components, the constructor is a special method used to initialize the component—for example, to set up state or bind methods. When you define a constructor and receive props, you must call super(props) as the first line to properly initialize the parent class (React.Component) and make this.props accessible. The super keyword in JavaScript calls the constructor of the parent class, ensuring the component is correctly set up before you add your own logic. If you don’t call super(props), trying to access this or this.props will cause errors.

Use this.state to create state and this.setState to update the state variables.

16/6/2025 – Life cycle of class based components.

[React lifecycle methods diagram](https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/)





useEffect vs componentDidMount()



27/6/2025

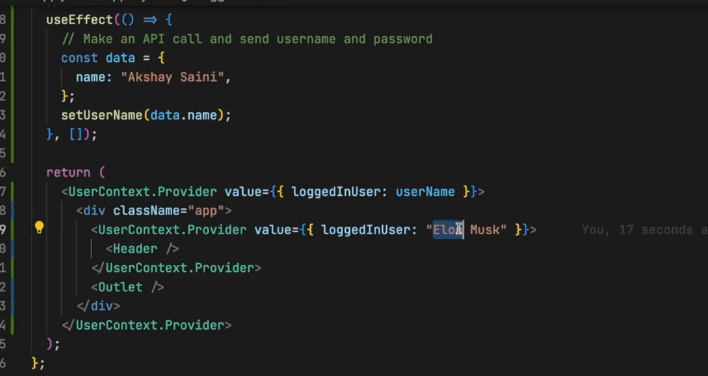
Tailwind css

The “rc” in . prettierrc stands for "runtime configuration"

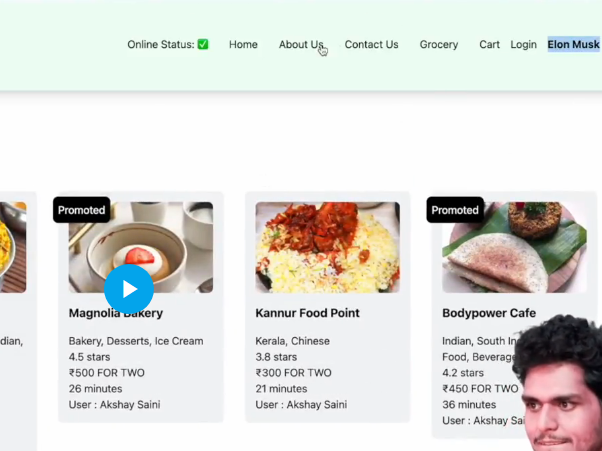
05/07/2025

Using the provider in different ways

Here we are wrapping only our header component with the loggedInUser value of “Elon musk”, so in our header the username will be Elon Musk. In the rest of our application we are wrapping it with username, and the value of useName is “Akshay Saini”. Therefore in the entire application other than header, loggedInUser will be ”Akshay Saini”.

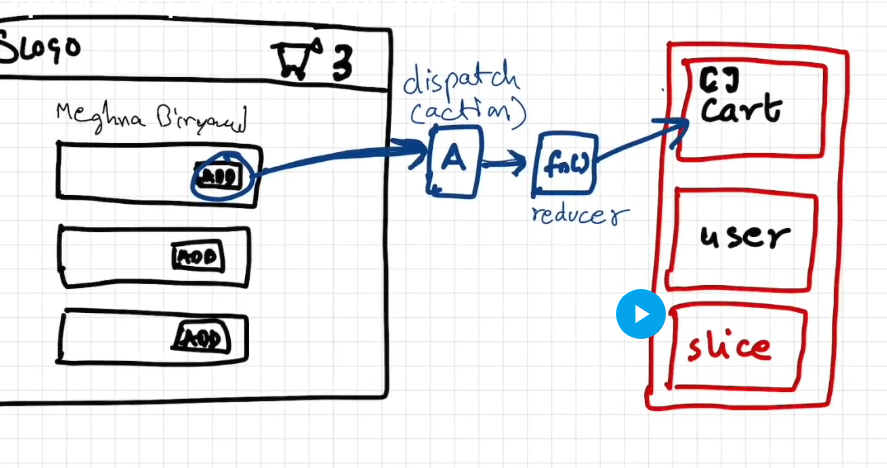


Output reference- see header and see the restaurantcard User.



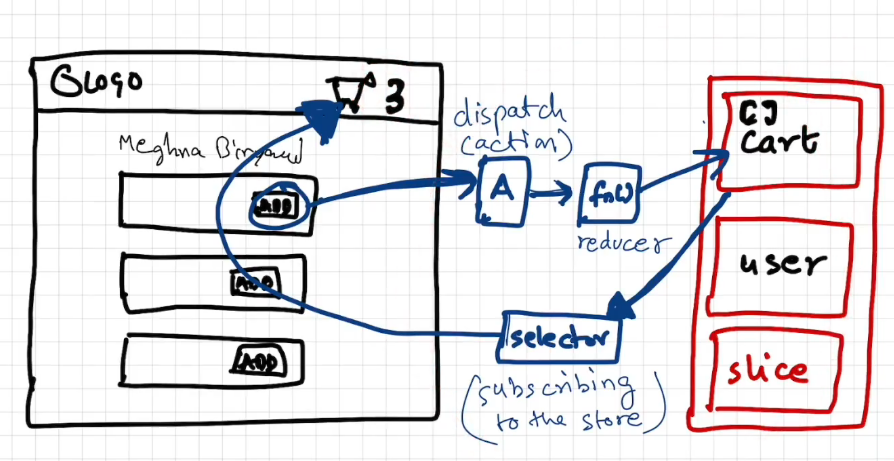
7/7/2025

Redux working architecture - When we click on the add button, it dispatches an action, which calls a reducer function which modifies the cart slice in the redux store.



When we have to read data from the store into a component – our component needs to subscribe to the store (reading data from the selector). The selector will read data from the store and use it in our component.

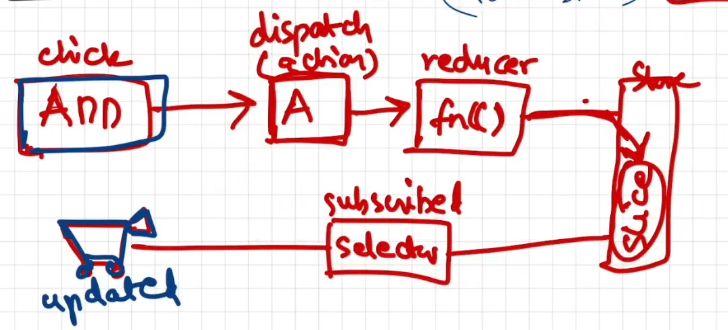
If a component has subscribed to the store, it means that it is in sync with the store.



A component is subscribed to the store using a selector.

Full Redux basic working





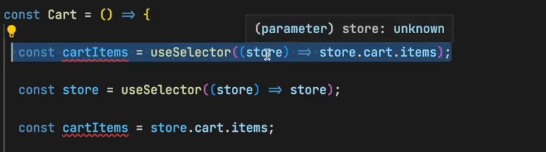
10/7/2025

Basic JS principle

| **Syntax** | **When it's called** | **Use when...** |
| --- | --- | --- |
| onClick={handleAddItem} | On click | No arguments needed |
| onClick={() => handleAddItem(item)} | On click | You need to pass arguments |
| onClick={handleAddItem(item)} | Immediately on render | ⚠️ Avoid this unless intentional |

Avoid using the last method as it will execute the function immediately on render.

**Important for interview preparation**



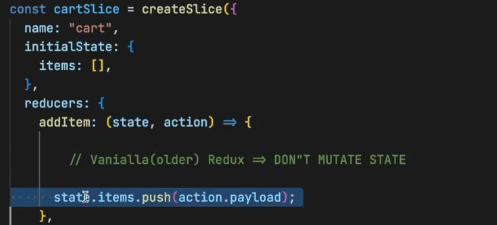
In the first line we are subscribing to only a part of our store which Is really required.

In the 2nd line we are subscribing to our entire store. This can cause a huge performance imbalance as subscribing to the entire store (which is a very large object) might be unnecessary, as we require only the items in this case.

In the second case whenever anything changes in the entire store the our Cart component will be updated. This is not required, we only want to subscribe to the updates in the ‘cart’ slice in the current scenario.

Only subscribing to a specific portion of the store will be more efficient and make our app more performant.

**Imp –** Earlier in older redux the below was not allowed, state was not allowed to be mutated

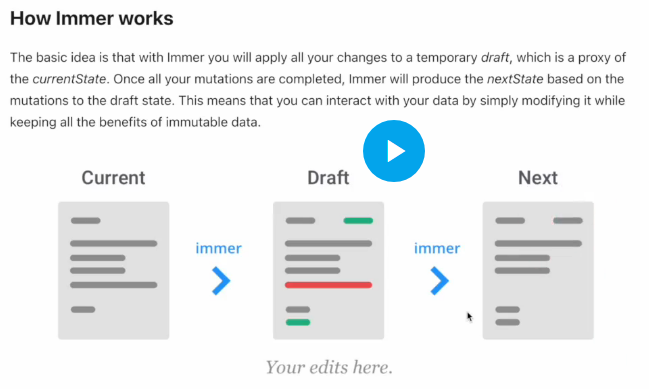
****

To modify state we would have to create a new state variable and modify the new state and then return this state.

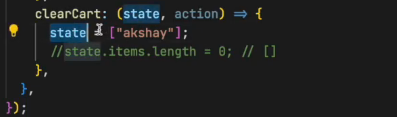


Redux behind the scenes is still doing whatever developers used to do in vanilla redux. But now with the incoming of RTK, developers don’t have to do this.

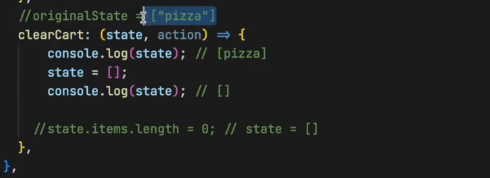
Redux does this with the help of a tiny package known as immer-



Here we are not mutating the state but just adding a reference to the state-

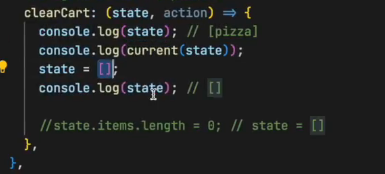


When we do the below (state=[]), we are not modifying the original state, we are just modifying the local variable in the scope of the function, the actual state is not changing.



Hence the original state is untouched and we do not see any of the changes being reflected in the UI.

**Imp**



When you're working inside a Redux Toolkit **reducer**, the state is a **Proxy object** (thanks to **Immer**). This allows you to write "mutating" code that actually produces immutable updates under the hood.

**🔍 However:**

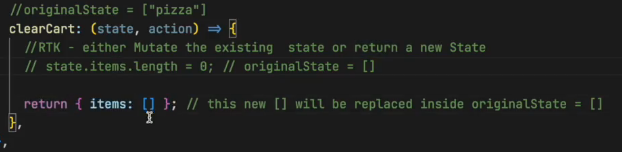
If you try to directly console.log(state), you'll see a **Proxy** object, not the plain JavaScript object you're used to — which makes it hard to read or debug the actual state data.

**✅ To safely inspect or access the real state:**

Use current(state) from @reduxjs/toolkit:



**Imp**

****

Here we are returning [], which is the new state. The commented-out line is the part which is mutating the existing state.

13/7/2025

Testing

React Testing Library (RTL) is a lightweight solution for testing React components that builds upon the core DOM Testing Library. Its primary goal is to encourage better testing practices by focusing on how users interact with your software, rather than implementation details.

Key aspects of React Testing Library and its relationship with JSDOM:

* **User-Centric Testing:**

RTL emphasizes testing from the user's perspective. This means interacting with and asserting on DOM elements in a way that mimics real user behavior, such as finding elements by their label text, button text, or accessible roles.

* **DOM Interaction:**

Instead of dealing directly with instances of rendered React components, your tests will work with actual DOM nodes. This is facilitated by utilities that allow querying the DOM in a user-like manner.

* **JSDOM Integration:**

When running tests in a Node.js environment (like with Jest), RTL leverages JSDOM to simulate a browser-like DOM environment. This allows your tests to interact with a virtual DOM, even though they are not running in a real browser.

* **Dependencies:**

To use React Testing Library, you typically install @testing-library/react along with its peer dependency @testing-library/dom. If you are using Jest, you will also commonly install @testing-library/jest-dom for extended DOM matchers.

* **Focus on Accessibility:**

By encouraging queries based on user-visible attributes like labels and text, RTL naturally promotes writing more accessible applications.

* **Avoids Implementation Details:**

RTL discourages testing internal component state, lifecycle methods, or private methods. The focus remains on the rendered output and user interactions.

In essence, React Testing Library provides a set of utilities that make it easy to write tests for your React components by interacting with the simulated DOM provided by JSDOM, ensuring your tests reflect how users will experience your application.

Jest supports multiple test file naming conventions by default. It will recognize any files inside a \_\_tests\_\_ directory (e.g., \_\_tests\_\_/filename.js) as test files, even if they don’t include .test or .spec in the filename. Additionally, it also looks for files named with .test.js, .spec.js, .test.tsx, etc., regardless of their folder location. This is controlled by the testMatch configuration in jest.config.js, which includes patterns like \*\*/\_\_tests\_\_/\*\*/\*.[jt]s?(x) and \*\*/?(\*.)+(spec|test).[tj]s?(x). While it’s not necessary to add .test when using \_\_tests\_\_ folders, many teams do so for consistency and clarity.

16/7/2025

**📝 Mocking fetch() in JSDOM Environment**

In a testing environment like jsdom, the fetch() API is not available by default because it's a browser-specific Web API. To simulate network requests in unit tests, we **mock the fetch() function manually**.

This mock involves **two levels of awaiting**:

1. **First await**: Simulates receiving a Response object from the fetch() call.
2. **Second await**: Simulates calling .json() on the response object, which itself returns a promise that resolves to the actual JSON data.

By mocking both these layers, we can reliably test components that rely on data fetching without depending on a real API call.

Additional React Notes

**⚛️ Client-Side Rendering (CSR) vs Server-Side Rendering (SSR) — Notes for React Interviews**

**✅ Client-Side Rendering (CSR)**

**Definition**:  
CSR is when the **browser** is responsible for rendering the entire UI using JavaScript after the initial HTML page loads.

**Flow**:

1. Browser requests a page.
2. Server responds with:
   * A minimal **HTML skeleton** (<div id="root"></div>)
   * A **JavaScript bundle** (React app + logic)
3. Browser executes the JS:
   * **Renders components**
   * **Fetches data via APIs**
   * **Attaches interactivity**

**Pros**:

* Rich interactivity (ideal for SPAs).
* Lower server load — only serves static assets & APIs.
* Good for internal tools/dashboards.

**Cons**:

* **Slower initial load** (especially on slow networks/devices).
* **Poor SEO** out-of-the-box (bots see empty HTML).
* JS failures can leave a blank page.

**Use Cases**:

* SPAs (e.g., dashboards, admin panels).
* Apps where SEO is not critical.

**✅ Server-Side Rendering (SSR)**

**Definition**:  
SSR is when the **server renders the full HTML** of a page, and the browser only handles hydration for interactivity.

**Flow**:

1. Browser requests a page.
2. Server:
   * Runs React (or templating engine)
   * Fetches data and renders HTML
3. Browser:
   * Receives and displays full HTML
   * **Loads JS to hydrate** components (make them interactive)

**Pros**:

* **Faster initial load** (content is visible immediately).
* **Better SEO** — crawlers can index full content.
* Useful for dynamic content that changes per request.

**Cons**:

* Heavier server load.
* More complex architecture (hydration, data fetching).
* Slower time to interactivity (due to hydration delay).

**Use Cases**:

* Marketing pages, blogs, e-commerce, content-heavy sites.
* Apps where SEO and performance matter.

**🔁 Key Differences**

| **Feature** | **CSR** | **SSR** |
| --- | --- | --- |
| Rendering Location | Browser | Server |
| Initial Load Speed | Slower | Faster |
| SEO Support | Poor (without extra config) | Excellent |
| Server Load | Lower | Higher |
| Interactivity | Immediate | After hydration |
| Data Fetching | On client (via APIs) | On server |
| Use Case | SPAs, dashboards | SEO-critical, dynamic content |

**🧠 Hybrid Approach**

* Frameworks like **Next.js** allow mixing:
  + **SSR** for SEO-heavy pages
  + **CSR** for client-only routes
  + **SSG** (Static Site Generation) for fast-loading static pages
  + **ISR** (Incremental Static Regeneration) for dynamic/static hybrids

**💧 Hydration in React**

**Hydration** is the process by which React **attaches event listeners and initializes component state** on an HTML page that was already rendered on the **server**.

**🧠 Think of it like this:**

The server sends **fully rendered HTML** to the browser. The browser **displays it immediately**, but it's just static HTML.  
Then React’s **JavaScript takes over**, connects the HTML to React’s virtual DOM, and makes it **interactive** — that’s hydration.

**✅ Steps of Hydration**

1. The server renders the React components and sends the **full HTML** to the browser.
2. The browser **displays the HTML** instantly — good for fast load & SEO.
3. The browser **loads the JS bundle**.
4. React **matches the existing HTML with its virtual DOM**, then:
   * Attaches event listeners (e.g., onClick, onChange)
   * Initializes state and hooks
   * Resumes interactivity

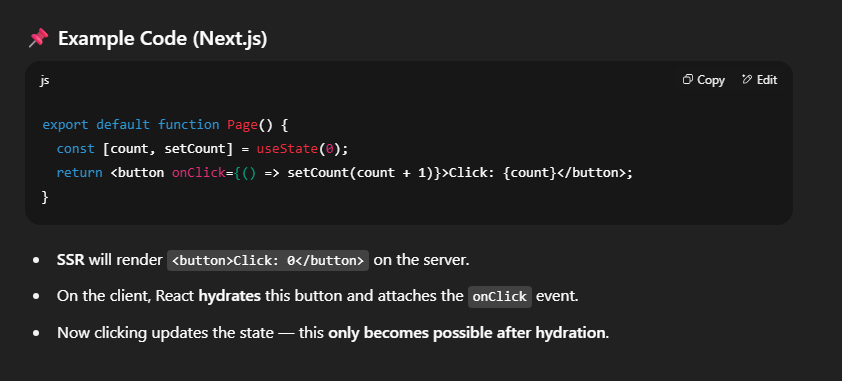
**🔄 Why Not Just Re-render on Client?**

Because:

* The HTML is **already rendered** — re-rendering would be wasteful.
* Hydration allows React to **"pick up where it left off"** from the server-side render.

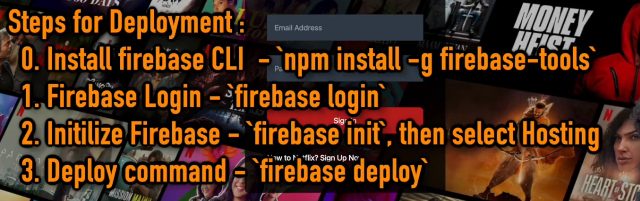
**⚠️ Common Issues During Hydration**

* **Mismatch** between server-rendered HTML and client-side React render leads to **hydration warnings**.
* For example: using random IDs, dates, or client-only state during server rendering can cause issues.



28/7/25

Deploying to firebase



**Unmounting & Mounting vs Re-rendering in React**

**1. Mounting**

* Happens **when a component is created and inserted into the DOM** for the first time.
* Functional components: function runs, then useEffect(..., []) runs.
* Class components: constructor → render → componentDidMount.
* **State starts fresh**.

**2. Unmounting**

* Happens **when a component is removed from the DOM**.
* Component instance is destroyed; **state is lost**.
* Cleanup runs:
  + Class: componentWillUnmount
  + Functional: cleanup function in useEffect.
* Example:

jsx

CopyEdit

{show && <MyComponent />} // When show = false → unmounts

**3. Re-rendering**

* Happens **when a component updates its output due to state, prop, or context change**, without being removed from the DOM.
* **Same component instance** is reused; state is preserved.
* Functional: function body runs again; class: render() runs again.
* useEffect(..., []) does **not** run again.
* Example:

jsx

CopyEdit

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

<button onClick={() => setCount(c => c + 1)}>+</button>

**Key Differences Table**

| **Aspect** | **Mount / Unmount** | **Re-render** |
| --- | --- | --- |
| Component instance | Created / Destroyed | Reused |
| State | Reset (lost) | Preserved |
| useEffect([]) run | Yes (on mount) | No |
| Cleanup | On unmount | No |
| DOM change | Insert / Remove | Update in place |

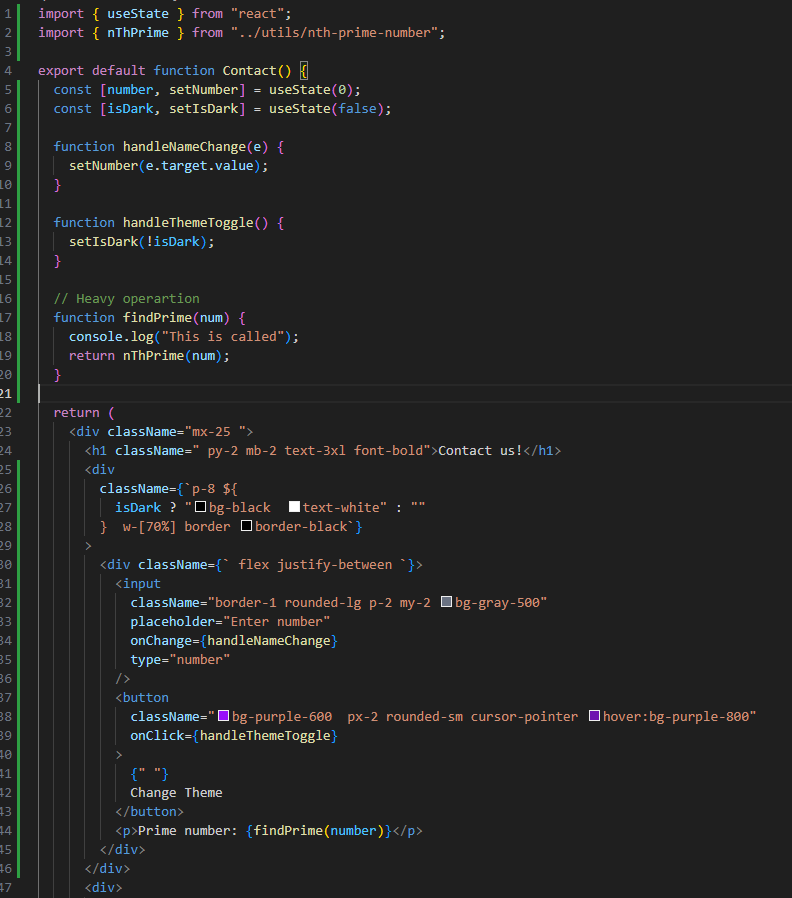
**Quick Analogy**

* **Mount/Unmount** → Turning a device on/off (new session each time).
* **Re-render** → Refreshing the screen without turning it off.

If you want, I can also add a **short code demo** showing logs for both cases so you can visualize exactly when mounting/unmounting vs re-rendering happens. That would make this even easier to remember.

15/8/2025

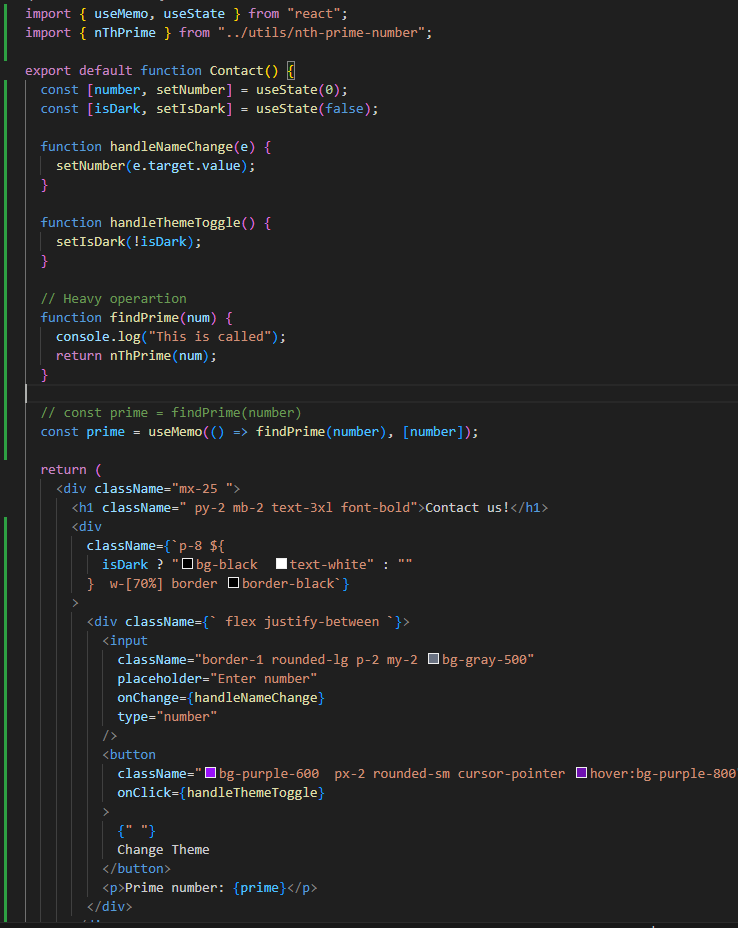
useMemo() Hook-



This is problematic code, because the findPrime function is being called on every render of the component. Even when the theme is being toggled we are calling the findPrime because that is how react reconciliation will work..

To prevent this we can use useMemo. useMemo takes in a callback function which will run initially and cache the result. Only when the value of the dependencies change the function will run again and cache the next new value.

In the context of the above example, we will pass the find prime into useMemo and add number as the dependency, so that when the number in the input box changes, only then we will re calculate the prime number. Otherwise use the cached result.



Below Is the code changes.

**React.memo()**

React.memo is a **higher-order component (HOC)** that lets you tell React:  
*"Only re-render this component if its* ***props actually change****."*

It’s like wrapping your component in a little “shouldComponentUpdate” check without writing the whole thing yourself.

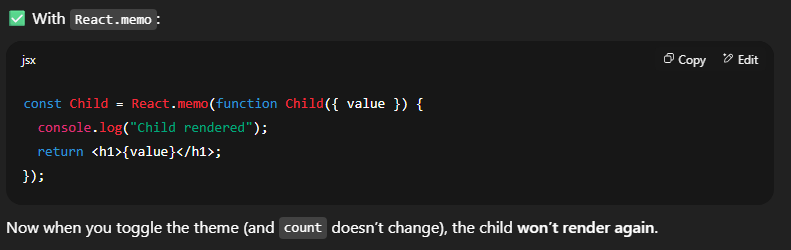
**How it works**

By default, when a parent re-renders, all its children also re-render, even if their props didn’t change.

React.memo wraps a functional component so that React **skips rendering** it if the props are *shallowly equal* to the last render.



Here the parent re-renders whenever there is a change in the theme, and if the parent re-renders, by default all the child components are also re-rendered. By using React.memo, we are preventing child re-renders unless the props of the child actually changed. Here in the above example we want to re-render the child if and only if the count changes, not for the theme change. Hence use, React.memo-



**When it works well**

* Child component renders **the same output** for the same props.
* Props are **primitive values** (strings, numbers, booleans) or **memoized objects/functions** (so their reference stays the same).
* You have **performance problems** due to unnecessary renders.

**What is useCallback?**

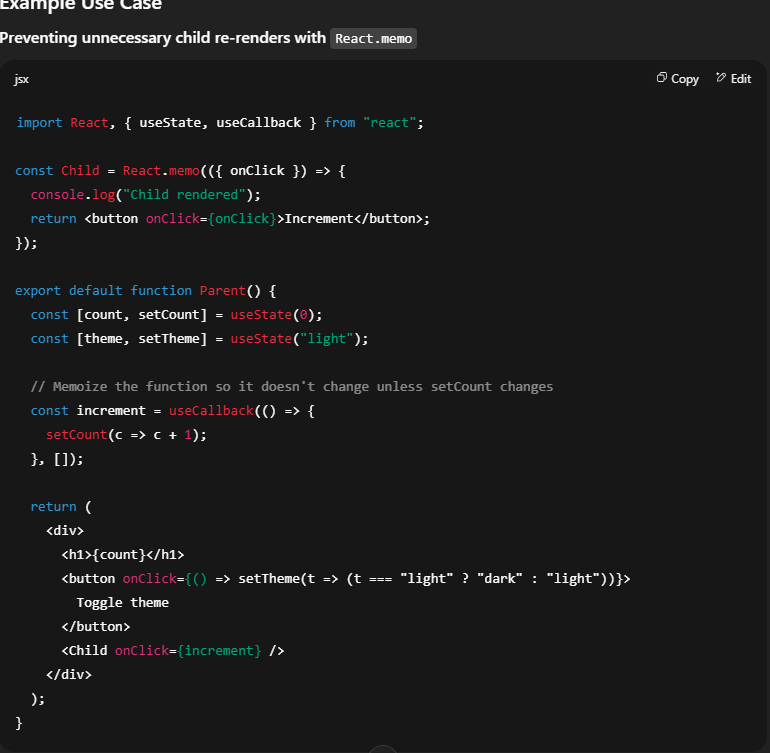
useCallback is a React Hook that **returns a memoized version of a function** —  
meaning the same function reference will be reused between renders **unless** its dependencies change.

**Why it exists**

In JavaScript, functions are objects — and every time a component re-renders,  
a brand-new function object is created even if the function body is identical.

This is usually fine, **but** it can cause:

1. **Unnecessary re-renders** in child components wrapped in React.memo.
2. **Repeated effect execution** when the function is in a useEffect dependency array.



**What happens here**

* Without useCallback:
  + Every parent re-render → new increment function reference → React.memo sees a prop change → child re-renders unnecessarily.
* With useCallback:
  + increment keeps the **same reference** across renders (as long as [] dependencies don’t change) → child doesn’t re-render unless increment actually changes.

💡 **Rule of thumb**:  
Use useCallback when:

* You pass a function to a memoized child (React.memo).
* The function is in a dependency array of useEffect / useMemo and you want to avoid triggering them unnecessarily.

`

Yes — React state variables are **independent of the JavaScript call stack** and the function’s **execution context**.  
That’s exactly how React can preserve your state between renders even though your component function runs from scratch each time.

Let’s unpack it step by step.

**1️⃣ What normally happens in plain JavaScript**

If you have:

function counter() {

let count = 0;

count++;

console.log(count);

}

counter(); // 1

counter(); // 1 again

Every call creates a **new execution context** → all variables inside are re-initialized → values reset.

**2️⃣ What happens in React components**

In React:

function Counter() {

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

return <button onClick={() => setCount(count + 1)}>{count}</button>;

}

Even though Counter() runs again on every re-render, useState does **not** just create a brand-new count each time.

Instead:

* React stores the state **outside** your component function, in an internal data structure called the **fiber tree**.
* When your component re-renders, React **looks up** the stored state value (based on the hook’s call order) and gives it back to you.
* The count variable inside your component is **just a local copy** of that stored value for this render.

**3️⃣ Why this works**

Think of it like this:

* **JavaScript call stack**: handles function execution and local variables → all gone after the function finishes.
* **React state store (fiber)**: lives *outside* the call stack → persists between renders.

So on the first render:

Fiber stores: count = 0

On click:

* setCount(1) schedules a re-render.
* React re-runs Counter() and passes 1 from fiber storage into count.
* Your JSX shows 1 on the screen.

**4️⃣ Visual analogy**

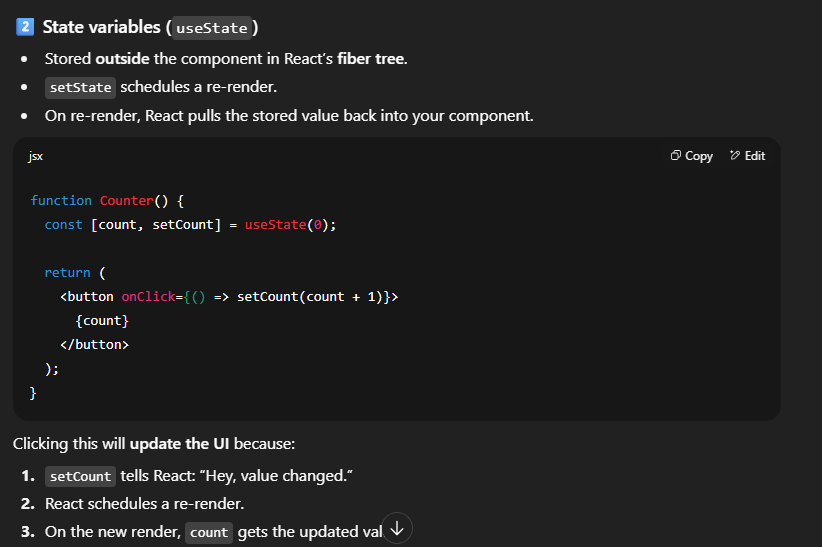
Imagine your component function is a **kitchen** and useState is a **fridge in another room**:

* Every time you cook (render), you **walk to the fridge** (React fiber) to get ingredients (state).
* Even if you leave the kitchen and come back (function ends and is re-run), the fridge keeps the ingredients from last time.

**5️⃣ Why hook order matters**

Since React matches state to hooks by **call order**, changing hook order between renders will confuse React —  
it won’t know which “fridge shelf” corresponds to which useState.





**💡 Why React doesn’t track normal variables**

React **is not a reactive system like Vue or Svelte** where variable mutations can trigger updates automatically.  
Instead, it’s a “pull” model — you **explicitly tell** React to update by calling a state updater (setState).

This design keeps React’s mental model predictable:

* **Local vars** → for temporary values that don’t affect rendering.
* **State vars** → for values that should trigger UI updates.

But if we want these let variables (non state variables) not to be reset when the functional component re-renders, we use useRef().