Monolith architecture and micro service architecture:

Separation of concerns and single responsibility principle

Monolith architecture is a architecture where all the features and services are coupled with the single codebase. Monolithic architecture works with a single executable file or directory which eases development. It is also easier to maintain because it uses less components. Testing and debugging operations are less intensive. In monolith architecture all the code is stored in a single codebase in one central location. The downside of monolith architecture is that it resistance to new technologies and it has reduced scalability.

Also always remember useEffect hook takes two arguments, first is the callback function and the second is the dependency. useEffect with an empty dependency will get called once react renders all the components on to the screen.

In our tiny example, the body component renders first and then after rendering is done it then calls the useEffect hook.

Shimmer UI – this is fake UI that is shown to the User until the actual data shows up on to the screen.

The reason why we try to use a state variable in react is to update the UI. If we were to update a variable name from one thing to another and show it in the UI it won’t work unless or until we use a state variable and use it to change the variable value only then it will actually be reflected in the UI which the user can see.

For example:

let loginBtn = “Login”

<button onClick={() => loginBtn=”Logout” console.log(loginBtn)}>{loginBtn}</button>

Now when coding this you will see that loginBtn is actually changing to Logout in the console but in the UI which we are trying to show it won’t update or won’t get reflected by this change. So for it to reflect in the UI we need to use a state variable instead.

const [loginBtn, setLoginBtn] = useState(“Login”);

<button onClick={() => setLoginBtn((btn) => btn === “Login” ? “Logout” : “Login”)}>{loginBtn}</button>

Now this code above will get reflected in the actual UI.

Sometimes when we update something using state variable in react we will see in the developer’s console only that element being getting updated and not the whole component itself, but that is not true. If you do a console.log(“Some message”) in the component itself and try to see you will see that the console.log statement gets printed twice. First for the initial render and then 2nd for the updated render. Only then you will be able to notice that it actually re renders the whole component and not just that element.

Whenever state variable updates, react triggers a reconciliation cycle (re-renders the component)

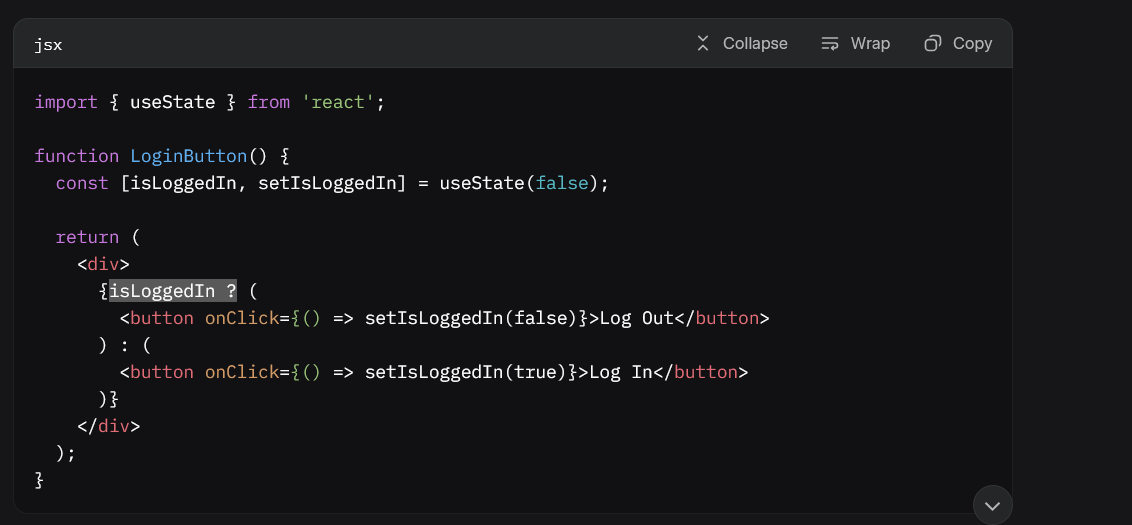
1. What is a Microservice?
2. Microservice architecture is a way to build services that are independent of each other and are loosely coupled. Each service is often owned by a single team or a small team of 10 to 5 people. It basically splits an app into smaller, independent pieces, where each piece handles one specific job. This principle is called separation of concerns and Single Responsibility Principle. Each individual service is interconnected with each other.
3. Simpler development: Microservices breaks down complex applications into smaller, easier-to-handle services. This makes development faster and easier to maintain.
4. Independent Teams: Each service can be developed independently by a team focused on that specific task.
5. Flexibility in Technology: Developers have the freedom to choose the best technology for that service, without being tied to choices made at the start of the project.
6. Continuous Deployment: Microservices allow for independent deployment, enabling continuous deployment for complex applications.
7. Scalability: Each services can be scaled independently, ensuring efficient resource usage.
8. Separation of Concerns: Since each task is it’s own independent project, the architecture stays organized and manageable.
9. What is Monolithic architecture?
   1. Monolithic architecture is when all parts of the application are bundled together in a single project. Everything is tightly connected and stored in one place.
   2. Some advantages of Monolithic architecture are:
      1. Simplicity of Development.
      2. Simplicity of debugging.
      3. Simplicity of testing.
      4. Simplicity of deployment.
      5. Simplicity of application evolution.
      6. Simplicity of onboarding new team members.
      7. Cross-cutting concerns and customizations are used only once.
   3. Some disadvantages of monolithic architecture includes:
      1. Slow speed of development: The simplest disadvantage relates to CI/CD pipeline. If the monolith contains a lot of services then each service is covered with tests that are executed for each pull request.
      2. High Code coupling
      3. Code ownership cannot be used
      4. Testing becomes harder
      5. Performance issues
      6. The cost of infrastructure
10. What is the difference between Monolith and Microservice?
    1. Monolith Architecture
       1. All features are built and run together in one big application
       2. One codebase, one deployment
       3. Easy to start, but hard to scale and manage as it grows.
    2. Microservice Architecture:
       1. The app is split into small, independent services (like login, billing, orders).
       2. Each service has its own codebase, can be deployed separately, and scaled on its own.
       3. More flexible, but harder to set up and maintain.
11. Why do we need a useEffect Hook?
    1. The useEffect hook is needed because React components are mainly about showing the UI (like buttons or text on the screen), but sometimes you need to do extra tasks that aren’t directly about the UI, like fetching data, setting timers, or updating the page title. These tasks often need to happen at specific moments, like when the component first shows up or when data changes. Without useEffect, there’s no built-in way in React to handle these tasks properly or at the right time. useEffect is a tool that lets us interact with the outside world but not affect the rendering or performance of the component that it’s in.
    2. The reason why useEffect is important:
       1. It handles extra tasks: Some tasks, like fetching data from a server or starting a timer, don’t fit into the normal flow of showing the UI. useEffect helps these tasks to be executed after the component has been rendered on to the screen.
       2. Controls timing: React component re-render or render when state or props change, but some tasks need to wait until after the screen updates. useEffect runs after the component is drawn, so it’s perfect for tasks like fetching data or updating the browser’s title.
       3. Cleans up Mess: Some tasks, like timers or subscriptions, keep running even if the component goes away (e.g. you navigate to another page). useEffect lets you “clean up” by stopping these tasks to avoid problems like memory leaks.
       4. Keeps Code Organized: Without useEffect, we would mix up UI with other tasks like data fetching making the code messy. useEffect keeps tasks separate and clear.
       5. For example:
       6. Import {useEffect, useState} from ‘react’;
       7. Function UserProfile(){
          1. const [user, setUser] = useState(null);
          2. useEffect(() => {
             1. fetch(<https://some.api.com/user>)
             2. .then(response => response.json())
             3. .then(data => setUser(data));
          3. return () => {
             1. console.log(“Cleanup: Stopping any ongoing fetch!”);
          4. };

}, []);

Return <div>{user ? user.name : “Loading…”}</div>;

* 1. Why useEffect here?
  2. Fetching data: since we are getting data from the server it takes time. useEffect runs this after the component shows “Loading…” on the screen.
  3. Timing: It waits until the UI is shown before fetching, so the app doesn’t freeze.
  4. Cleanup: If the user navigates away before the data arrives, the cleanup function can stop the fetch to avoid errors.
  5. **Why Not Without useEffect?**
  6. If you put the fetch code directly in the component, it would run every time the component render, causing multiple unnecessary fetches or errors.
  7. Without useEffect, you’d have no way to clean up tasks like timers or subscriptions, which could keep running and cause bugs.
  8. useEffect ensures these tasks happen at the right time and only when needed, keeping the app smooth and bug-free.
  9. **How Does useEffect Work?**
  10. It takes two parts:
      1. A function: The task you want React to do, like fetching data or logging a message.
      2. A Dependency list: A list that tells React when to run the task, like when certain values change.
  11. After react draws the component on the screen (called “rendering”), it checks useEffect and runs the function if needed.
  12. QUESTION: If useEffect with an empty dependency array ([]) runs only once after the component shows up, will it run every time I refresh the page? And if I have a state variable that changes on a button click, causing the component to re-render, will useEffect run on each re-render? If so, what’s the point of the empty dependency array?
  13. Answer:
  14. **What Happens with an Empty Dependency Array ([])?**
      1. The empty dependency array means the useEffect will run only once after the component is first shown on the screen (this is called the initial render).
      2. This is useful for tasks that we want only once, like fetching data when the component loads or setting up a one-time configuration (e.g. updating the page title).
  15. **Does useEffect Run on Page Refresh?**
      1. Yes, if refreshed, the entire React app reloads, and the component is treated as if it’s being added to the page for the first time.
      2. A page refresh is like starting the app from scratch, so React rebuilds the component, and the useEffect runs as part of the initial render.
  16. **What Happens When a State Variable Changes (e.g., on Button Click)?**
      1. When clicked a button that changes a state variable (e.g. increasing a counter), React re-renders the component to update the UI with the new state.
      2. However, if the useEffect has an empty dependency array ([]), it does not run again during these re-renders. The empty array means React ignores all state changes and only runs the useEffect code when the component is first added to the screen.
      3. This is different from a useEffect with a specific dependency (like [count] for example), which would run on every render or when the listed dependency changes.
      4. If we don’t specify an empty array dependency then the useEffect would run after every render, which could cause problems like fetching data repeatedly or starting multiple timers, slowing down the app or creating bugs.

1. What is Optional Chaining?
   1. The optional chaining (?.) operator accesses an object’s property or calls a function. If the object accessed or function called using this operator is undefined or null, the expression short circuits and evaluates to undefined instead of throwing an error.
   2. Syntax:
      1. obj.prop
      2. obj?.[expr]
      3. func?.(args)
   3. The ?. operator is like the . chaining operator, except that instead of causing an error if a reference is nullish (null or undefined), the expression short-circuits with a return value of undefined. When used with function calls, it returns undefined if the given function does not exist.
2. What is Shimmer UI?
   1. Shimmer UI is a fake interface shown to users while the real data is loading. It keeps the user experience smooth by displaying a temporary layout (like placeholders for images or text) until the actual content is ready.
3. What is the difference between JS expression and JS statement?
   1. A JavaScript expression produces a value, while a JavaScript statement performs an action.
   2. Definition: An expression is a piece of code that evaluates to a single value. IT can be used anywhere a value is expected.
   3. Definition: A piece of code that does something, like declaring a variable or running a loop, but doesn’t always produce a value.
   4. The main key difference: expressions give you a value (e.g. 5 + 3 = 8). Statements do actions (e.g. if or let). In JSX, curly braces {} hold expressions because they need values, not statements.
   5. A screenshot of a computer program

      AI-generated content may be incorrect.
4. What is Conditional Rendering, explain with a code example?
   1. Conditional Rendering in React is when you show or hide parts of the UI based on certain conditions, like whether a user is logged in or a value is true/false.
   2. A simple example is showing a “log in” button for guests or a “Log Out” button for logged-in users.
   3. 
   4. State: isLogginIn tracks whether the user is logged in (true or false).
   5. Condition: The ternary operator (isLoggedIn ? … : …) checks isLoggedIn:
      1. If true, shows a “Log Out” button
      2. If false, shows a “Log In” button
5. What is CORS?
   1. CORS stands for Cross-Origin Resource Sharing. It’s a security feature in web browsers that controls how a website on one domain (or address) can request resources (like data, images, or scripts) from a different domain
   2. When you visit a website (say, www.myapp.com), it might try to fetch data from another website (like www.api.com). Browsers have a security rule called the Same-Origin Policy that says a website can only access resources from its own domain (same address) unless the other website explicitly allows it. This prevents malicious websites from stealing data from others.
   3. CORS is the system that lets servers (like www.api.com) say, “It’s okay for www.myapp.com to access my data.” It uses special instructions (called HTTP headers) to tell the browser whether the request is allowed or blocked.
   4. **WHY DO NEED CORS?**
   5. Security: Without CORS, any website could secretly request data from another site, which could lead to stealing user information (like login details) or performing harmful actions.
   6. Controlled Sharing: CORS lets servers decide which websites can access their resources, making it safe to share data with trusted sites.
   7. Common Use Case: For example, if your React app on www.myapp.com needs to fetch user data from an API on www.api.com, CORS ensures this happens only if www.api.com allows it.
   8. **HOW DOES CORS WORK?**
   9. **Request from Browser:** 
      1. When your website (www.myapp.com) sends a request to another domain (www.api.com), the browser adds an Origin header to the request (e.g., Origin: www.myapp.com).
      2. This tells the server where the request is coming from.
   10. Server Response:
       1. The server checks the Origin and decides if it’s allowed to share data.
       2. It sends back a special header called Access-Control-Allow-Origin to tell the browser what’s allowed. For example:
       3. Access-Control-Allow-Origin: www.myapp.com means only www.myapp.com can access the data.
       4. Access-Control-Allow-Origin: \* means any website can access it (less secure, used for public APIs).
   11. If the server doesn’t allow the origin, the browser blocks the response, and your app can’t see the data.
   12. ERRORS:
       1. If the server doesn’t allow the origin (e.g., missing or incorrect Access-Control-Allow-Origin), the browser shows a CORS error in the console, like: No 'Access-Control-Allow-Origin' header is present on the requested resource.
   13. Key points to remember:
       1. CORS is browser based
       2. Not all requests need CORS: Requests to the same domain (e.g. www.myapp.com fetching from [www.myapp.com/api](http://www.myapp.com/api)) don’t trigger CORS.
       3. Common Headers to remember:
          1. Access-Control-Allow-Origin: Specifies allowed domains
          2. Access-Control-Allow-Methods: Lists allowed HTTP methods (e.g. GET, POST).
          3. Access-Control-Allow-Headers: Lists allowed custom headers.
       4. Why errors happen: If the server doesn’t send the right CORS headers, or your app’s domain isn’t allowed, the browser blocks the response.
6. What is async and await?
   1. What is async?
      1. The async keyword is used to declare functions as asynchronous
      2. An async function always returns a promise, even if we don’t explicitly specify it or return a promise from the function
      3. Inside an async function, we can use the await keyword to pause execution of the function until the promise gets resolved.
      4. Async functions can contain one or more await
   2. What is await?
      1. The await keyword can only be used inside an async function. That means the function should have an async keyword in front of it.
      2. It tells javascript to wait for a Promise to resolve (finish) before moving to the next line of code.
   3. To handle errors we use try/catch to catch errors if the promise rejects.
   4. 
   5. Some key points to remember are:
      1. Async Function: Always returns a Promise. Use async to allow await inside.
      2. Await: Pauses execution until a Promise resolves, but only inside an async function.
      3. Use try/catch to handle rejected promises
      4. While await pauses the function, javascript can still handle other tasks such as user clicks because it is non-blocking.
   6. Some common pitfalls to watch out for:
      1. Don’t use await outside of an async function (it will cause syntax error).
      2. Don’t forget to handle errors using try/catch block.
      3. Avoid calling async functions directly in useEffect without wrapping them, as useEffect doesn’t expect a Promise. Instead define the async function separately and then call it inside the useEffect.
      4. 
7. What is the use of `const json = await data.json();` in getRestaurants()
   1. The reason why we are using await data.json() the response object’s json method takes time to show the data and convert it into json and that’s the reason why we are using await in front of the data.json.
   2. data.json() returns a Promise, and await is used to get the resolved value
   3. Correct explanation: The line const json = await data.json(); in the fetchData() function is used to parse the raw response from the fetch request into a JavaScript object.