



React Client-Components

Outline

1. Server vs Client Components
2. Component State
3. Components Communication
4. Common React Hooks
5. Interleaving Server and Client Components

Server vs Client Components



CLIENT COMPONENT

v/s



SERVER COMPONENT

Server vs Client Components in Next.js

- **Server Components**

- Run on the server. Great for:
 - Data fetching (from DB, API)
 - Static/SEO-friendly pages
 - Keep sensitive information on the server (access tokens, API keys, etc.)
 - Layouts and heavy logic



Use server components by default for performance and smaller bundle size (reduced client-side JavaScript)



Cannot use browser APIs or hooks such as `useState`, and `useEffect`

- **Client Components**

- Run in the browser: support React hooks like `useState`, `useEffect`
- While they execute on the client, **Next.js pre-renders their initial HTML** on the server for faster perceived load times and SEO
- Needed for:
 - Adding interactivity and event listeners (`onClick()`, `onChange()`, etc)
 - Animations, modals, dropdowns
 - Access to browser APIs (`localStorage`, `window`)



Use client components only when **interactivity is required**



Decision Tree

- Does the component use React hooks (e.g., useState/useEffect)?
 - └ Yes → Use Client Component
- Does it access browser-only APIs (e.g. window, localStorage)?
 - └ Yes → Use Client Component
- Does it fetch data from a Database or a Web API?
 - └ Yes → Use Server Component
- Is it interactive (e.g., handles client-side events such as click, mouseOver)?
 - └ Yes → Use Client Component



- **Default to Server Components** for performance and enhanced security
- **Isolate** interactivity into **small** Client Components where needed

Component State

$$f(\text{State}) =$$

State

name: John
surname: Dough

View

Component State

- A component can store its own local data (**state**)
 - Private and fully controlled by the component
 - Can be passed as **props** to children
- Use **useState** hook to create a *state variable* and an *associated function* to update the state

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

useState returns a state variable *count* initialized with 0 and a function *setCount* to be used to update it

- Calling *setCount* causes React to **re-render the app components** and **update the DOM** to reflect the state changes



Never change the state directly by assigning a value to the state variable => otherwise React will NOT re-render the UI

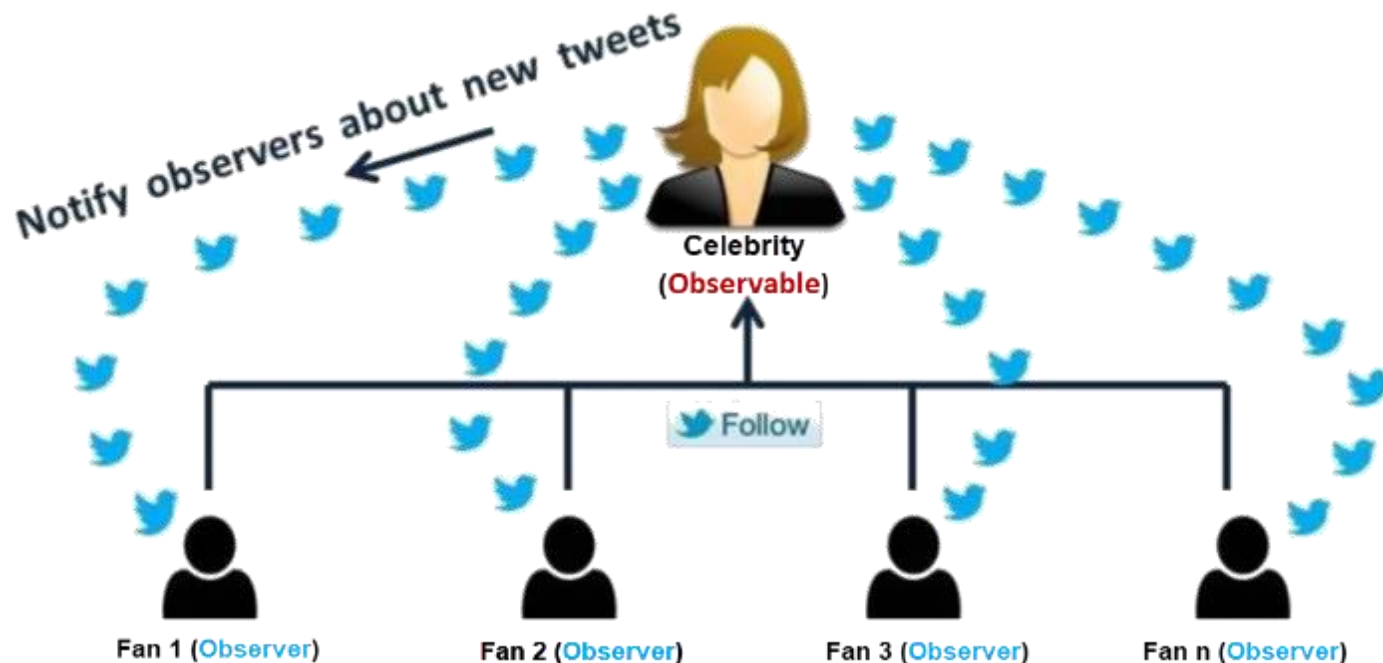
State

- State = any value that can change overtime
 - State variable must be declared using **useState** hook to act as **Change Notifiers**
 - They **are observed** by the React runtime
 - **Persist** their values i.e., remembers the previously stored value (e.g., count = 5) across re-renders
 - Any change of a state variable will trigger the **re-rendering** of any functions that **reads** the state variable
 - Both props and state changes trigger a render update
- => UI is **auto-updated** to reflect the updated app state

Observer Pattern at the heart of Jetpack Compose

Observer Pattern Real-Life Example: A celebrity who has many fans on Tweeter

- Fans want to get all the latest updates (posts and photos)
- Here fans are **Observers** and celebrity is an **Observable** (analogous **state variable** in React)
- A **State variable** is an **observable data holder**: React runtime **observes its changes** and updates the UI accordingly



Imperative UI vs. Declarative UI

- Imperative UI – manipulate DOM to change its internal state / UI

```
document.querySelector('#bulbImage').src = 'images/bulb-on.png';  
document.querySelector('#switchBtn').value = "Turn off";
```



UI in React is immutable

- In react you should NOT access/update UI elements directly (as done in the imperative approach)
- Instead update the UI is by updating the state variable(s) used by the UI elements – this triggers automatic UI update
 - E.g., change the bulb image by updating the *isBulbOn* state variable

```
<input type="button"  
  value= {isBulbOn ? "Turn off" : "Turn on"}  
  onClick={() => setIsBulbOn(!isBulbOn)} />
```

useState: creates a state variable

- Used for basic state management inside a component

const [state, setState] = useState(initialState)



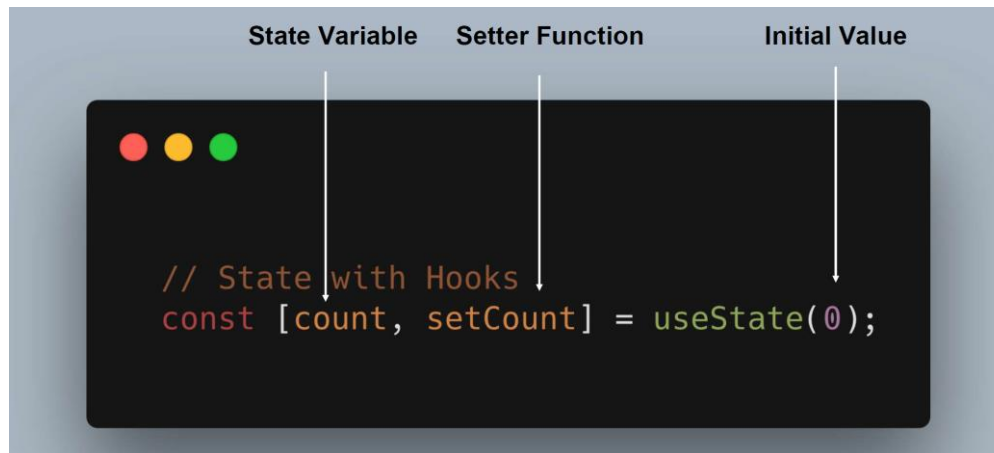
The name of
your state



The function you'll
eventually use to
change the value of this
state



The initial value
of your state



Component with State + Events Handling

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

Count: 4



```
function Counter({ startValue }) {  
  const [count, setCount] = useState(startValue);  
  const increment = () => { setCount(prev => prev + 1); };  
  
  const decrement = () => { setCount(prev => prev - 1); };  
  
  return <div>  
    Count: {count}  
    <button type="button" onClick={increment}>+</button>  
    <button type="button" onClick={decrement}>-</button>  
  </div>  
}  
export default Counter;
```



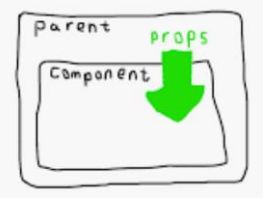
Handling events is done the way events are handled on DOM elements

- Use the **Counter** component

```
<Counter startValue={3}/>
```

Uni-directional Data Flow:

Props vs. State

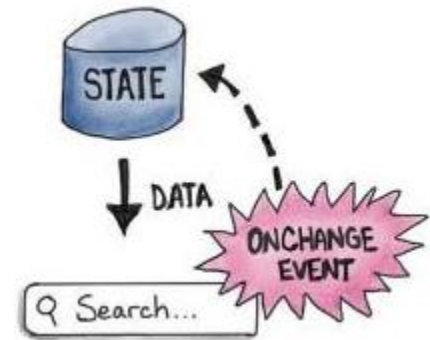
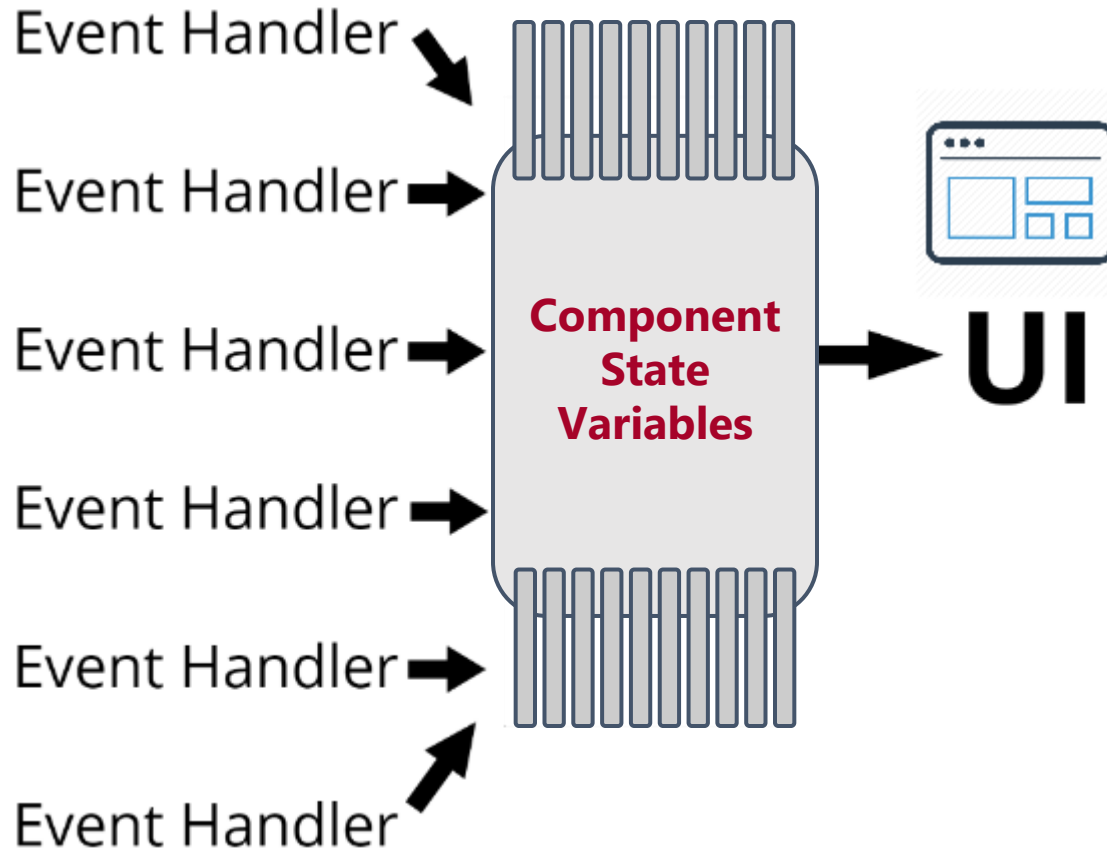


- **Props** = data passed to the child component from the parent component
- **Props** parameters are **read only**

- **State** = internal data managed by the component (cannot be accessed or modified outside of the component)
- **State** variables are **Private** and **Modifiable** inside the component only (through **set** functions returned by `useState`)

👍 React **automatically re-render the UI** whenever **state** or **props** are updated

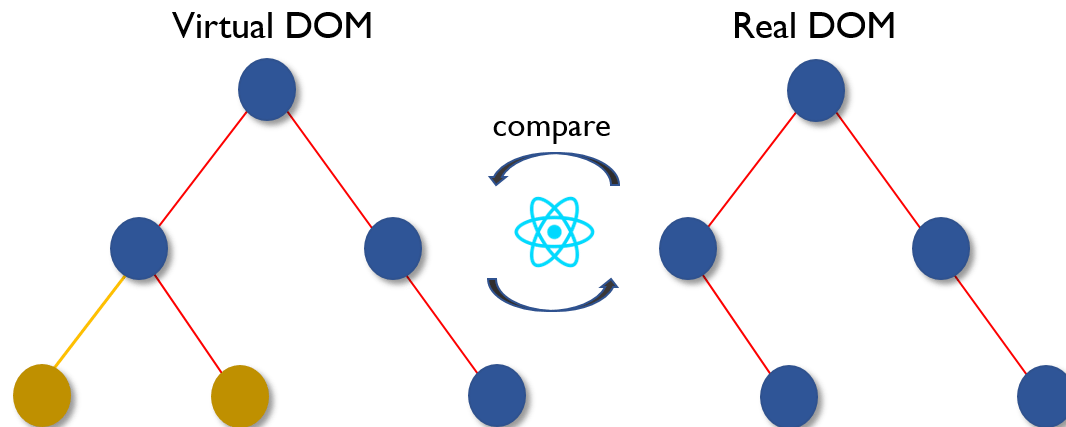
Event Handlers update the State and React updates the UI



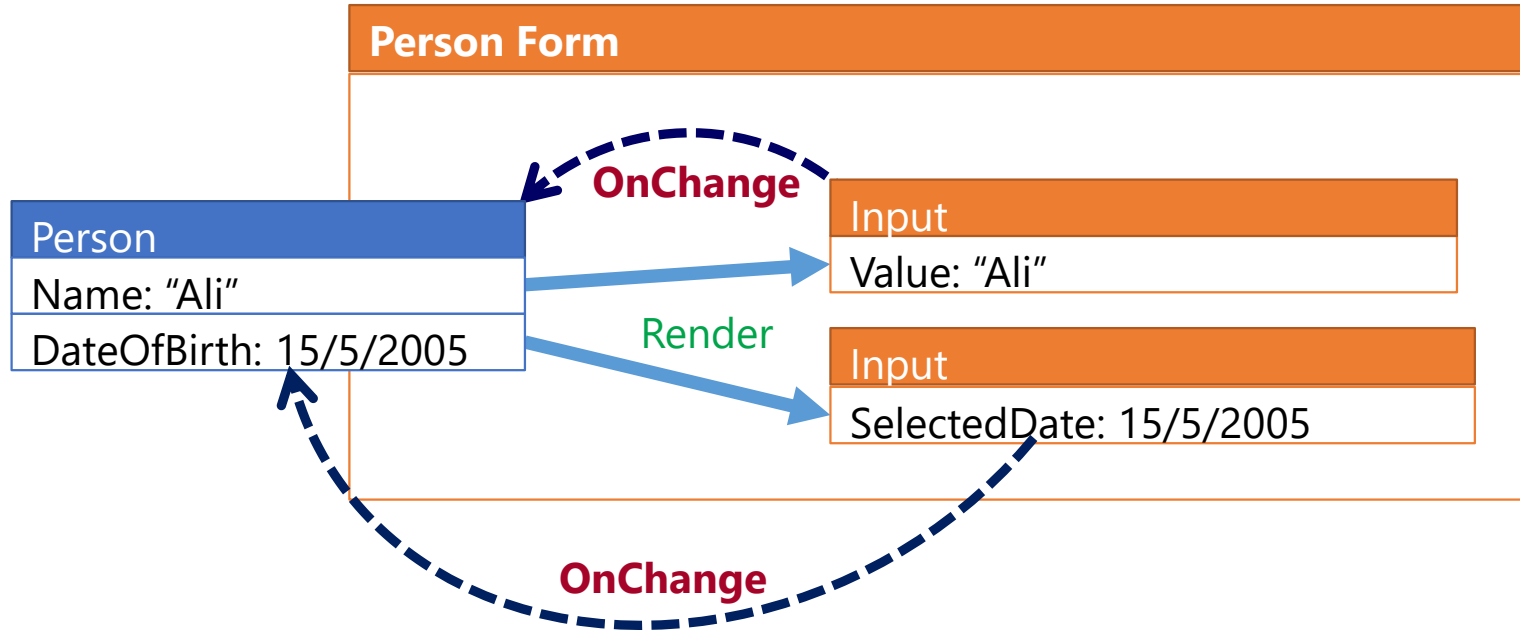
Every place a state variable is displayed is guaranteed to be auto-updated

Virtual DOM

- Virtual DOM = Pure JavaScript lightweight DOM, totally separate from the browser's slow JavaScript/C++ DOM API
- Every time the component **updates its state** or **receives new data via props**
 - A new virtual DOM tree is generated
 - New tree is **diffed** against old...
 - ...producing a minimum set of changes to be performed on real DOM to bring it up to date



Unidirectional Data Flow in Forms



Common Events: `onClick` - `onSubmit` - `onChange`

Forms with React

Form UI

```
<form onSubmit={handleSubmit}>
  <input
    name="email"
    type="email" required
    value={state.user}
    onChange={handleChange} />
  <input
    name="password"
    type="password" required
    value={state.password}
    onChange={handleChange} />
  <input type="submit" />
</form>
```

```
const [state, setState] = useState({ email: "", password: "" });
```

```
const handleChange = e => {
  const name = e.target.name;
  const value = e.target.value;
  //Merge the object before change with the updated property
  setState({ ...state, [name]: value });
};
```

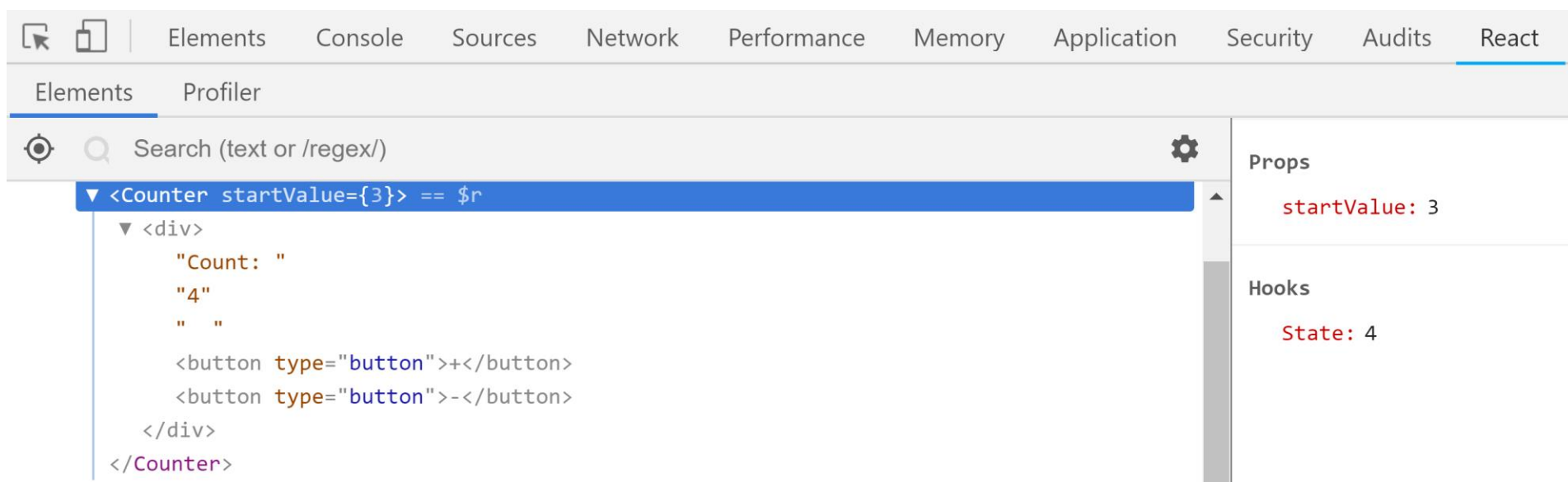
```
const handleSubmit = e => {
  e.preventDefault();
  alert(JSON.stringify(state));
};
```

Form State and Event Handlers

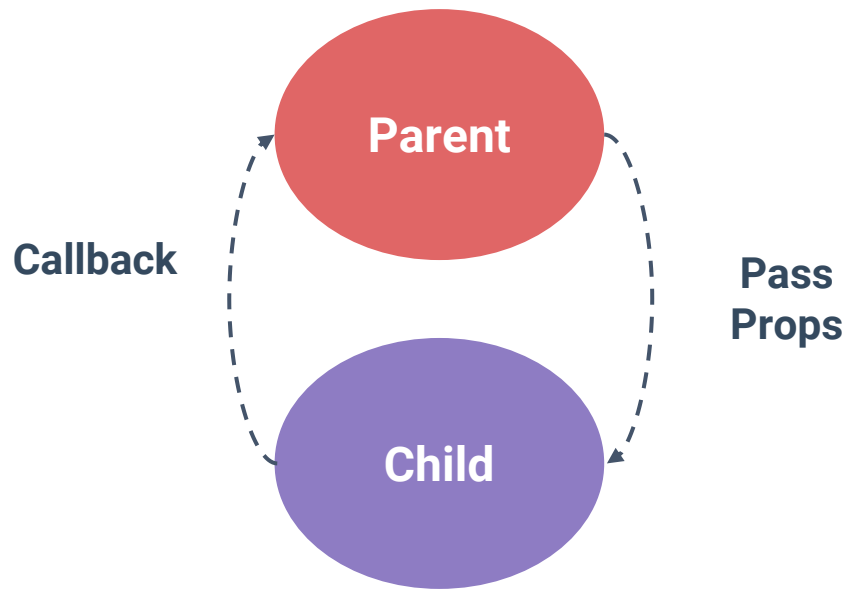
React Dev Tools

- React Dev Tools

<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en>

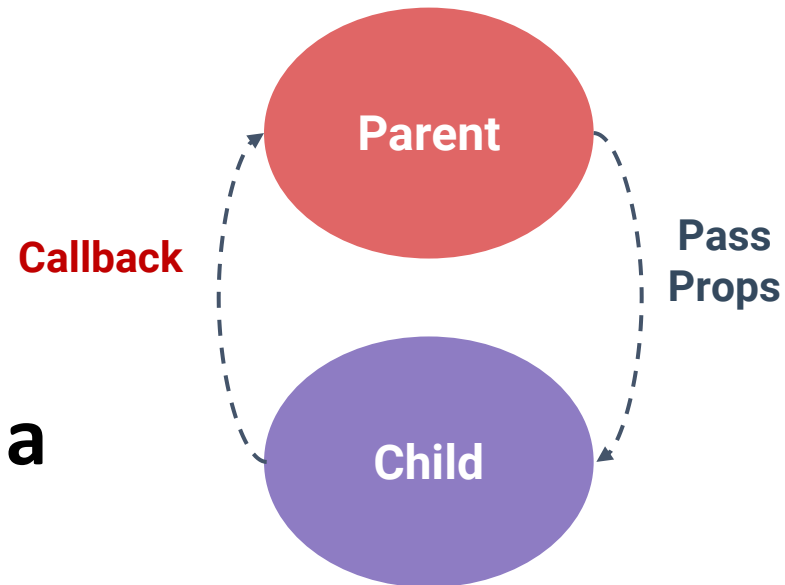


Components Communication

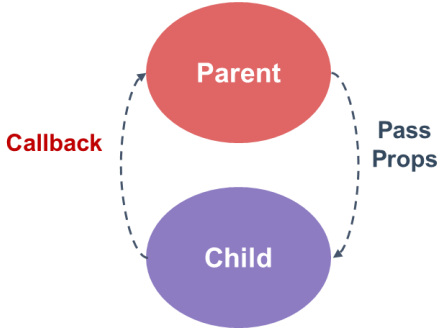


Composing Components

- Components are meant to be used together, most commonly in parent-child relationships
- Parent passes data down to the child via **props**
- The child notify its parent of a **state change via callbacks** (a parent must pass the child a callback as a parameter)



Parent-Child Communication



Parent

```
function Main => <Counter startValue={3}  
  onChange={count => console.log(`Count from the child component: ${count}`)} />
```

Child

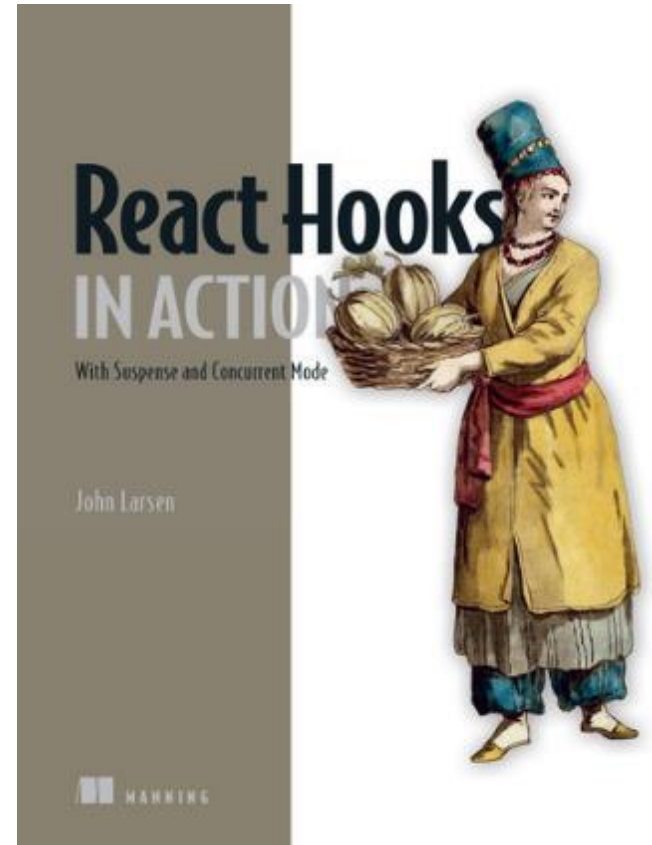
```
function Counter({startValue, onChange}) {  
  const [count, setCount] = useState(startValue);  
  
  const increment = () => {  
    setCount(prev => prev + 1);  
    onChange(count);  
  };  
  
  return <div>  
    Count: {count}  
    <button type="button" onClick={increment}>+</button>  
  </div>  
}
```

Common React Hooks



React Hooks

1. useState
2. useEffect
3. useRef
4. useContext
5. useRouter
6. useActionState
7. useOptimistic



Some of the slides are based on
<https://learning.oreilly.com/library/view/react-hooks-in/9781617297632/>

What is Hook?

- React Hooks: Functions enabling the use of React state and lifecycle features within functional components. E.g.,
 - `useState`: adds and manages local state within a component
 - `useEffect`: enables running code such as fetching data after a component mounts
- Rules of Hooks:
 - Call at Top Level: Only call Hooks at the very top level of the React function, not inside loops, conditions, or nested functions
 - Don't Call Conditionally: (Implicit in Rule 1) Avoid calling Hooks inside conditional (if/else) blocks

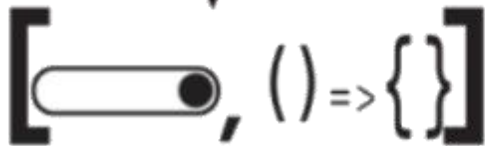
useEffect

- For doing stuff when a component is mounts/unmounts/updates
- Ideal for fetching data when the component is mounted

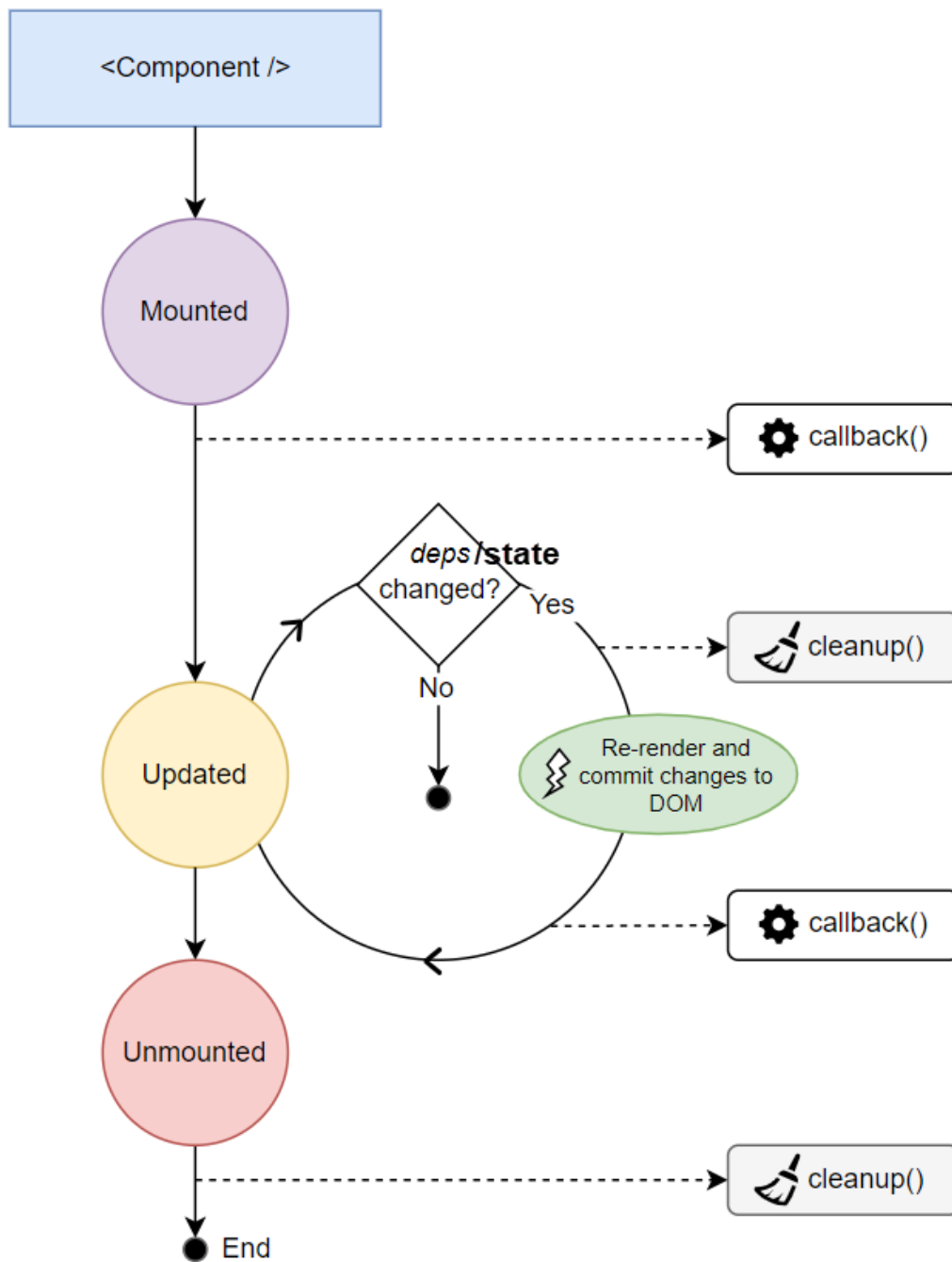
```
useEffect( () => {  
  // do something with dep1 and dep2  
  return () => { /* clean up */ };  
}, [dep1, dep2] );
```



Cleanup function:
Return a function to clean up
after the effect (e.g., unsubscribe,
stop timers, remove listeners, etc.).



Dependency list:
Run the effect only if the
values in the array change.



1) After Initial Render: React runs the effect callback function after the component first mounts

2) After Subsequent Renders (if dependencies change):

- React first runs the cleanup function (if provided)
- Then, React runs the effect callback function

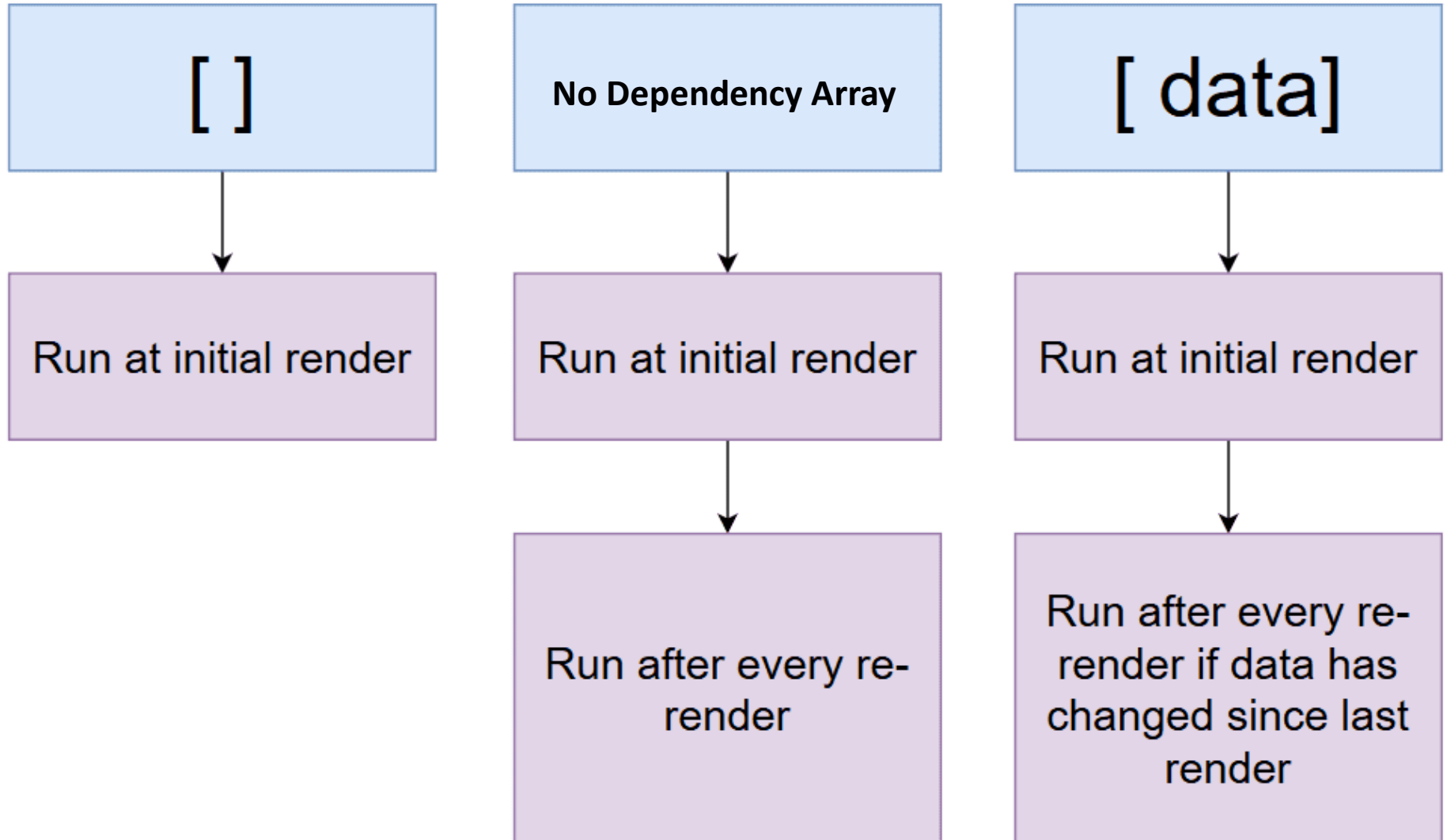
3) Before Unmount: React runs the cleanup function (if provided) just before the component is removed

Common usage scenarios of useEffect

Common usage of useEffect include:

- Fetching data when the component mounts or when specific props/state change
- Working with timers like `setInterval` or `setTimeout`
- Using APIs like Geolocation, and `localStorage`

useEffect - 2nd argument



Use cases for the useEffect hook

Call pattern	Code pattern	Execution pattern
No second argument	<pre>useEffect(() => { // perform effect });</pre>	Run after every render.
Empty array as second argument	<pre>useEffect(() => { // perform effect }, []);</pre>	Run once, when the component mounts.
Dependency array as second argument	<pre>useEffect(() => { // perform effect // that uses dep1 and dep2 }, [dep1, dep2]);</pre>	Run whenever a value in the dependency array changes.
Return a function	<pre>useEffect(() => { // perform effect return () => {/* clean-up */}; }, [dep1, dep2]);</pre>	React will run the cleanup function when the component unmounts and before rerunning the effect.

useEffect – Executes code during Component Life Cycle

- **Initialize state data** when the component loads

```
useEffect(() => {
  async function fetchData() {
    const url = "https://api.github.com/users";
    const response = await fetch(url);
    setUsers( await response.json() ); } // set users in state
  fetchData();
}, []); // pass empty array to run this effect once when the component is first mounted to the DOM.
```

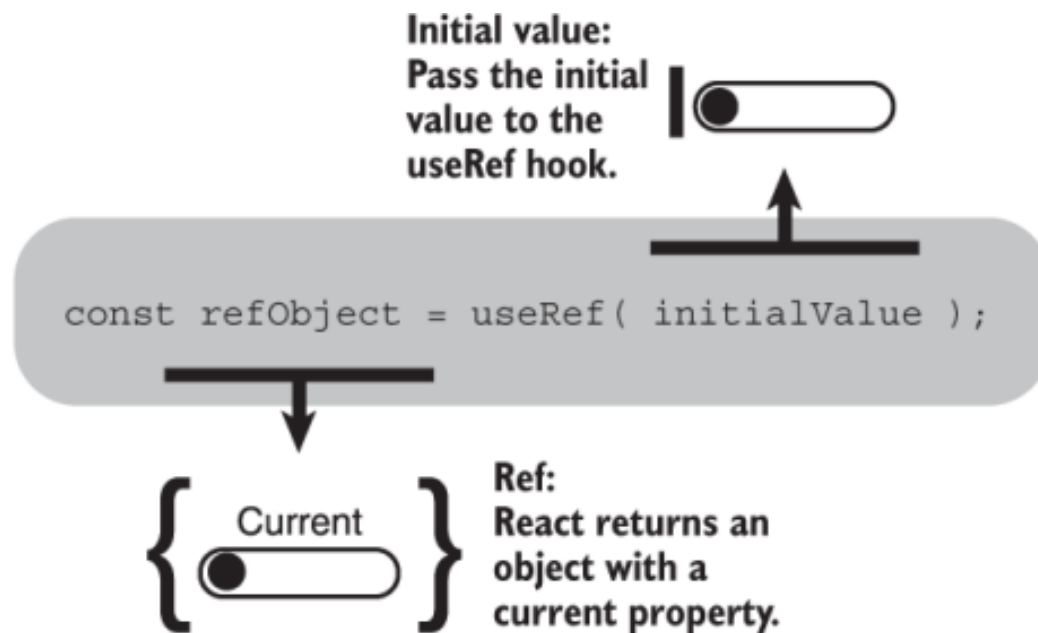
- **Executing a function every time a state variable changes**

```
useEffect(() => {
  async function fetchData() {
    const url = `https://hn.algolia.com/api/v1/search?query=${query}`;
    const response = await fetch(url);
    const data = await response.json();
    setNews(data.hits);
  }
  fetchData();
}, [query]);
```

If 2nd parameter is not set, then the useEffect function will run on every re-render

useRef

- useRef() hook to create **persisted mutable values** as well as directly **access DOM elements** (e.g., focusing an input)
 - The value of the reference is persisted (stays the same) between component re-renderings
 - Updating a reference doesn't trigger a component re-rendering.



useRef for Mutable values

- `useRef(initialValue)` accepts one argument as the initial value and returns a reference. A reference is an object having a special property `current`

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

function LogButtonClicks() {
  const countRef = useRef(0);

  const handle = () => {
    countRef.current++;
    console.log(`Clicked ${countRef.current} times`);
  };

  console.log('I rendered!');

  return <button onClick={handle}>Click me</button>;
}
```

- `reference.current` accesses the reference value, and `reference.current = newValue` updates the reference value
- The value of the reference is persisted (stays the same) between component re-renderings
- Updating a reference doesn't trigger a component re-rendering

useRef for accessing DOM elements

- useRef() hook can be used to access DOM elements

```
import { useRef, useEffect } from 'react';

function InputFocus() {
  const inputRef = useRef();

  useEffect(() => {
    inputRef.current.focus();
  }, []);

  return (
    <input
      ref={inputRef}
      type="text"
    />
  );
}
```

- Define the reference to access the element

```
const inputRef = useRef();
```

- Assign the reference to **ref** attribute of the element:

```
<input ref={inputRef} />
```

- After mounting, `inputRef.current` points to the DOM element

=> In this example, we access the input to focus on it when the component mounts. After mounting we call

```
inputRef.current.focus()
```

Store Previous State Value

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

export default function PreviousValueTracker() {
  const [count, setCount] = useState(0);
  const prevCountRef = useRef();

  useEffect(() => {
    prevCountRef.current = count; // update after render
  }, [count]);

  return (
    <div>
      <p>Current count: {count}</p>
      <p>Previous count: {prevCountRef.current ?? 'N/A'}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

prevCountRef stores the previous count value **after each render**. Unlike useState, updating useRef **does not cause a re-render**.

useRef vs. useState

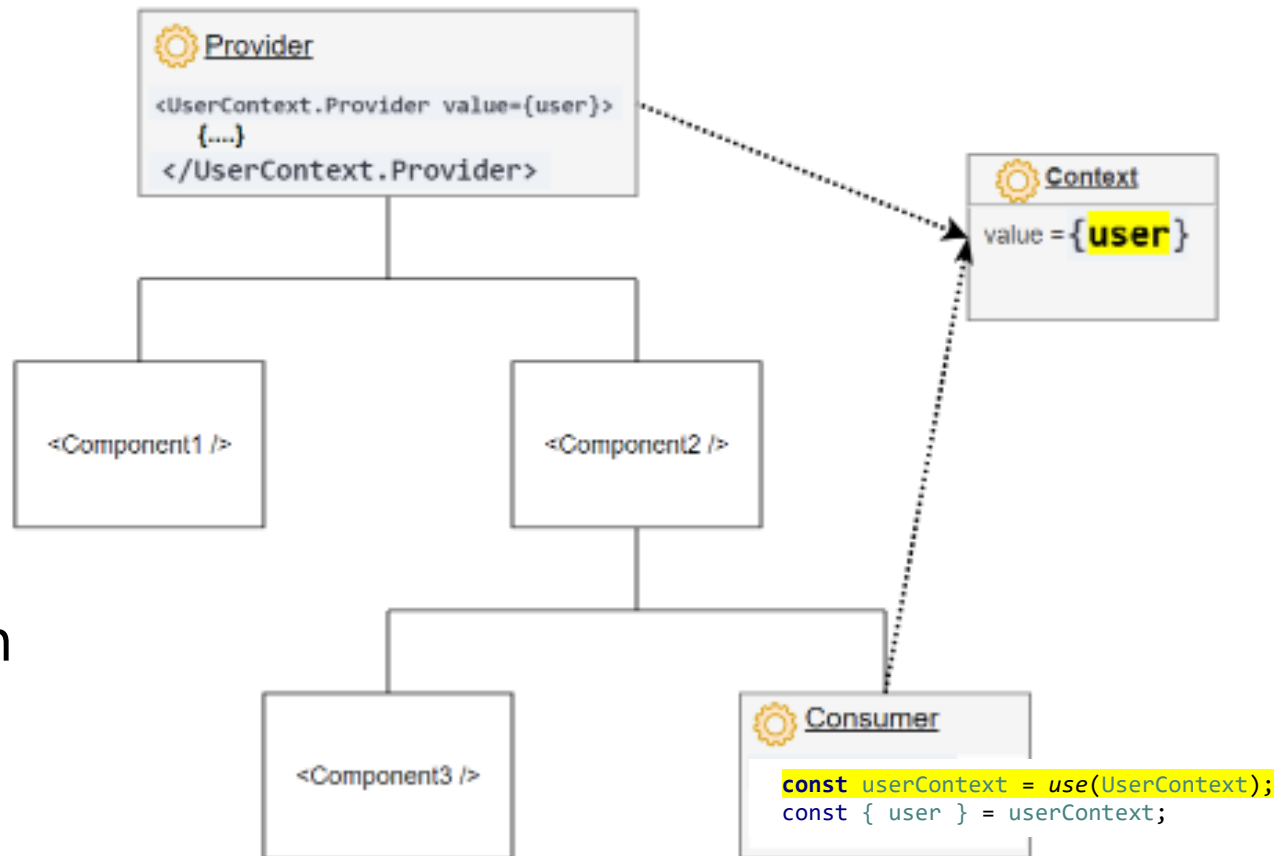
- useState, and useContext hooks triggering re-renders when a state variable changes
- useRef remembers the state value but change of value does not trigger rerender
 - The values of refs persist throughout render cycles

useContext

- Share state (e.g., current user, user settings) between deeply nested components more easily than prop drilling (i.e., without pass the state as props through each nested component)

- Using the context requires 3 steps:
creating, providing,
and **consuming** the context

- If the context variables change then all consumers are notified and **re-rendered**



useContext – provides shared variables and functions

1. **Create a context** instance (i.e., a container to hold shared variables and functions)

```
import {createContext } from 'react';  
const UserContext = createContext();  
export default UserContext;
```

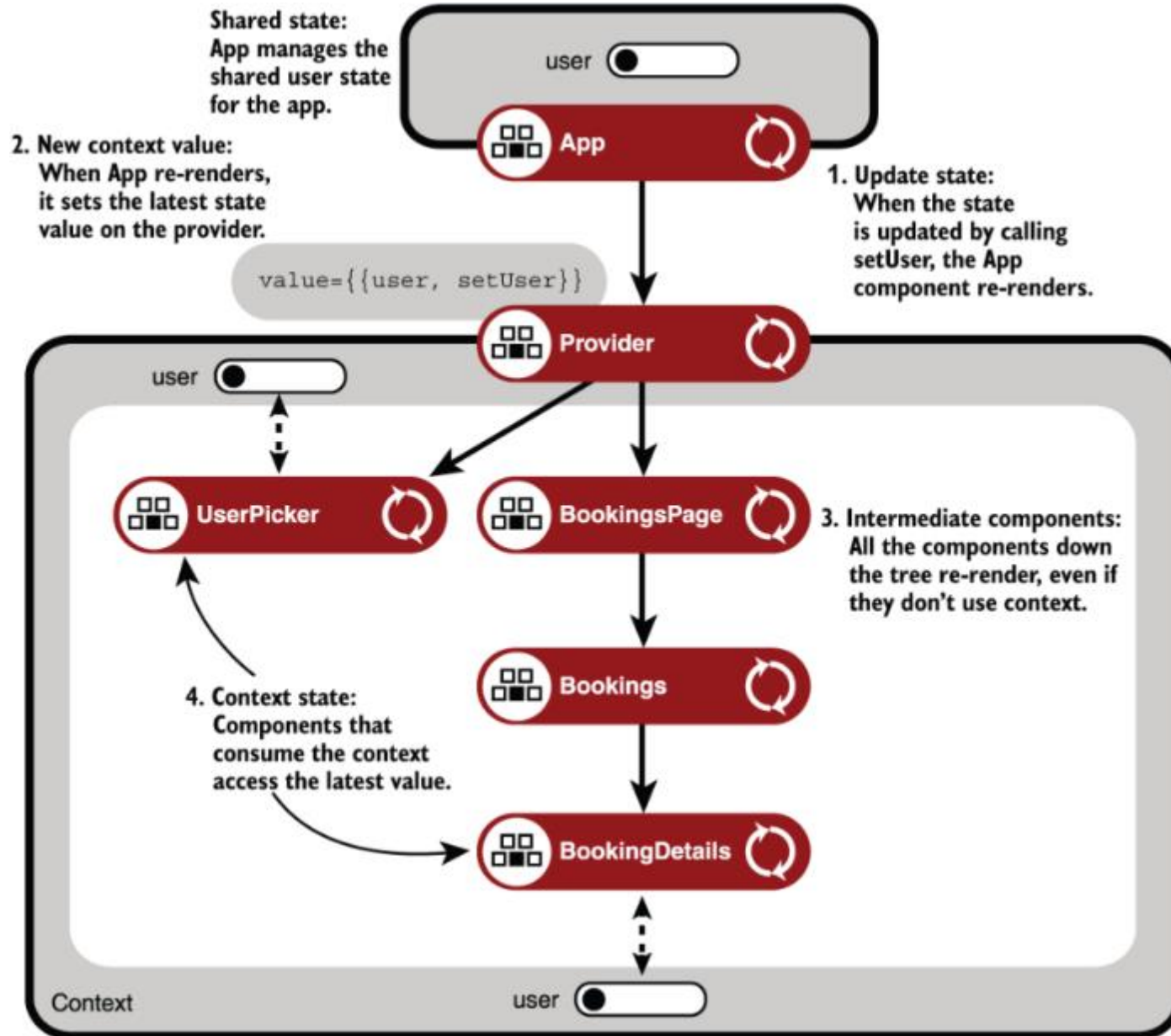
2. **Provider places shared variables / functions in the context** to make them available to child components

```
import UserContext from './components/UserContext';  
function App() { return (  
  <UserContext.Provider value={ user }>  
    <Welcome /> ...  
  </UserContext.Provider>  
); }
```

3. **Consumer access the shared variables / functions in the context**

```
import { use } from "react"; import UserContext from './UserContext';  
export default function Welcome() {  
  const userContext = use(UserContext);  
  const { user } = userContext;  
  return <div>You are login as: {user.username}</div>;  
}
```

Shared State Example



useRouter

- **useRouter** hook allows you to programmatically change routes inside Client Components
 - Use the <Link> component for navigation unless you have a specific requirement for using useRouter

```
'use client'

import { useRouter } from 'next/navigation'

export default function Page() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => router.push('/dashboard')}>
      Dashboard
    </button>
  )
}
```

useActionState

- **Purpose:** [useActionState](#), simplifies managing the lifecycle of a server action (pending, success, error) and the data returned by it - typically in response to form submissions
- **Signature:** `const [state, formAction, isPending] = useActionState(actionFn, initialState);`

Parameters:

- **actionFn:** An async Server Action that performs logic/validation and returns a new state object (success or error information)
- **initialState:** An object representing the initial form state (e.g., feedback messages, error status).

Returns:

- **state:** state: The latest state, updated based on actionFn's result
 - **formAction:** A function to trigger the actionFn. Typically assigned to a form's action attribute `<form action={formAction}>`
 - **isPending:** A boolean indicating whether the actionFn is currently executing
- **Benefits:** Automatically handles the pending UI state, simplifies updating the UI with success or error feedback after an action

Example

```
"use client";

import { useActionState } from 'react';
import { addTaskAction } from "../actions";

export default function TaskAdder() {
  const [state, formAction, isPending] = useActionState(addTaskAction, initialState );
  return (
    <div>
      <h2>Add New Task</h2>
      <form action={formAction}>
        <label htmlFor="taskInput">Task Name: </label>
        <input type="text" id="taskInput" name="taskName" disabled={isPending} required />
        <button type="submit" disabled={isPending}>
          {isPending ? 'Adding...' : 'Add Task'}
        </button>
      </form>

      { /* Display feedback based on the state message */ }
      {state.message !== null && (
        <p style={{ color: state.error ? 'red' : 'green', marginTop: '10px' }}>
          {state.message}
        </p>
      )}
    </div>
  );
}
```

Server Action

```
"use server";

// Define the asynchronous action function (returns new state)
async function addTaskAction(previousState, formData) {
  const taskName = formData.get('taskName');
  // Basic validation
  if (!taskName || taskName.trim().length < 3) {
    return { message: 'Task name must be at least 3 characters long.', error: true };
  }

  try {
    // Simulate network delay/database operation
    await new Promise(resolve => setTimeout(resolve, 1000));

    // Return the new state on success
    return { message: `Successfully added task: "${taskName}"`, error: false };
  } catch (error) {
    return { message: error.message || 'Failed to add task due to a server error.', error: true };
  }
}
```

useOptimistic

- **useOptimistic**: helps you optimistically update the UI before a server action completes
 - Improves user experience by providing feedback while awaiting confirmation from the backend
 - Once the server responds, the real state replaces the optimistic one
- **Signature:**

```
const [optimisticState, addOptimistic] = useOptimistic(actualState, updateFn);
```

Parameters:

- **actualState**: The current state from the server
- **updateFn**: A function used to compute a temporary state

Returns:

- **optimisticState**: The UI state including optimistic updates while waiting
- **addOptimistic**: A function you call to apply the optimistic update


Example

```
import { useState, useOptimistic } from 'react';
export default function CommentsPage() {
  const [comments, setComments] = useState([]);
  const [optimisticComments, addOptimisticComment] = useOptimistic(
    comments,
    (prevComments, newComment) => [...prevComments, newComment]
  );

  async function handleSubmit(formData) {
    const newComment = formData.get('comment');
    addOptimisticComment(newComment);
    const saved = await addCommentToServer(newComment);
    setComments(prev => [...prev, saved]);
  }

  return (
    <form action={handleSubmit}>
      <input type="text" name="comment" placeholder="Add a comment" required />
      <button type="submit">Post</button>
      <ul>
        {optimisticComments.map((c, i) => (
          <li key={i}>{c}</li>
        ))}
      </ul>
    </form>
  );
}
```

use Hook

- **Purpose:** The **use** hook lets a client component suspend rendering until a Promise (such as a Server Action) resolves
- **How it works:**
 - The component using **use(promise)** is typically wrapped in a `<Suspense>` component
 - While the Promise is pending, `<Suspense>` shows the fallback UI (e.g.,  Loading server data...)
 - Once the Promise resolves, the fetched content is replaces to fallback UI
- **Benefits:**
 - It makes asynchronous code look synchronous inside components, simplifying logic
 - While the Promise is pending, React `<Suspense>` handles showing the fallback UI without needing manual loading flag

use Hook Example

```
import { Suspense } from "react";
import UserName from "../components/UserName";
import { fetchUserName } from "../actions";

export default function Page() {
  const userPromise = fetchUserName(); // Call the Server Action
  return (
    <main>
      <Suspense fallback={<div>⌚ Loading user info...</div>}>
        <UserName userPromise = { userPromise } />
      </Suspense>
      <p style={{ marginTop: "20px" }}>Other content on the page.</p>
    </main>
  );
}
```

- This component will suspend while waiting for the server data
- At first, you see ⌚ Loading user info...
- Once the Promise is resolved, the fetched data is displayed, replacing the loading fallback

The **use** hook lets the client component suspend rendering until the promise (i.e., Server Action) resolves

```
"use client";
import { use } from "react";

export default function UserName({ userPromise }) {
  const user = use(userPromise);
  return <div>Hello, {user.name}!</div>;
}
```

Asynchronous Server action

```
"use server";
export async function fetchUserName() {
  await new Promise((resolve) => setTimeout(resolve, 2000));
  return { name: "John Doe" };
}
```

Summary of Hooks

- **useState:** Manage local component state
- **useEffect:** Handle side effects and respond to component lifecycle events
- **useRef:** Persist values across renders or directly reference DOM elements
- **useContext:** Share state/functions globally without prop drilling
- **useRouter:** Access routing information and navigation methods
- **useActionState:** Manage form state updates and errors when using server actions
- **useOptimistic:** Implement optimistic UI updates by temporarily updating state before a server action completes

Interleaving Server and Client Components



Interleaving Server and Client Components (1 of 2)

- "use client" is the Entry Point to Client-Side Interactivity
 - It marks where the server-rendering stops and client-side rendering begins for that component and anything it imports
 - The component and everything it imports are part of the client-side JavaScript bundle
- Best practice is to **keep Client Components Small**: Push interactivity as far down the component tree as possible (towards the "leaves")
 - Make as Client Component only specific elements or sections that require client-side interactivity or browser APIs
 - Avoid placing "use client" high up in the tree if only a small part needs it
 - This minimizes the JavaScript sent to the browser, leading to faster page loads and better performance
- **Server Components can import and render Client Components (SC → CC):**
 - It can pass props (strings, numbers, objects/arrays) down to the Client Component
 - This is the primary way to embed interactive components within server-rendered content

Interleaving Server and Client Components (2 of 2)

- **Client Components cannot directly import server components:**
 - Components marked with "use client" runs in the browser and cannot directly import server-components as they may contain server-only code, data fetching logic that require dependencies unavailable in the browser (like filesystem or database access)
- **Pass Server Components as children prop for composition (CC ← SC via Props):**



- This is a workaround to overcome the previous limitation
- The SC renders on the server, and the resulting content is slotted into the CC during the server render pass (i.e., when Next.js pre-renders the page initial HTML on the server)
- Ideal for **interactive layout components** (like Modals, Tabs, Accordions marked with "use client") that need to wrap and display complex content rendered on the server (e.g., data-heavy sections, components accessing server-only resources)
 - This keeps the content server-rendered while the container is interactive

Server Component Renders a Client Component (SC -> CC)

- **Use Case:** You fetch data and render content on the server, but need a specific interactive element within that structure (e.g., a button, a form, a dynamic counter)
- **Why:** Keep the bulk of the page rendered on the server for performance and SEO, delegating only the interactive parts to the client
- **Example:** A product page (SC) displays product details fetched from a database and includes an interactive "Add to Cart" button (CC)

Example: SC Renders a CC

```
// product/[id]/page.jsx (Server Component - Default)
import AddToCartButton from '../components/AddToCartButton'; // Client Component
import { getProductDetails } from '../lib/db'; // Server-side data fetching

export default async function ProductPage({ params }) {
  const product = await getProductDetails(params.id); // Fetches data on the server

  return (
    <div>
      <h1>{product.name}</h1>
      <p>{product.description}</p>
      <p>Price: ${product.price}</p>
      { /* SC renders the CC, passing necessary props */ }
      <AddToCartButton productId={product.id} productName={product.name} />
    </div>
  );
}
```

Client Component accepts Server Component via children Prop (CC <- SC as children)

- **Use Case:** You have an interactive wrapper component (like a modal, tabs, accordion, tooltip, or layout element needing client-side state) that needs to display content rendered on the server
- **Why:** Allows the interactive "shell" to manage its state/behavior on the client, while the content inside benefits from server rendering (faster initial load, potentially complex server-side logic/data fetching for the content)
- **Example:** An interactive <Accordion> (CC) that displays different sections of server-rendered content (SCs)

Example: CC accepts SC via children Prop (CC <- SC as children)

```
"use client";
import { useState } from "react";
import styles from "../styles.module.css";
export default function Accordion({ title, children, startOpen = false }) {
  const [isOpen, setIsOpen] = useState(startOpen);
  return (
    <div className={styles.accordion}>
      <button
        onClick={() => setIsOpen(!isOpen)}
        className={styles.accordionButton}
      >
        {isOpen ? ▼ : ► } {title}
      </button>
      {isOpen && <div className={styles.accordionContent}>{children}</div>}
    </div>
  );
}
```

► Readme.md (Server Rendered)

▼ Static Content Section

This is just static JSX defined within the parent server component.

```
import Accordion from "../components/Accordion";
import MarkdownContent from "../components/MarkdownContent";
export default function HomePage() {
  <Accordion title="Readme.md (Server Rendered)">
    </* This is a Server Component being passed as children */>
    <MarkdownContent contentFile="README.md" />
  </Accordion>
};
```

Resources

- Thinking in React

<https://react.dev/learn/thinking-in-react>

- Hooks

- <https://react.dev/reference/react/hooks>
- [React Hooks in Action textbook](#)

- Useful list of resources

<https://github.com/rehooks/awesome-react-hooks>

- Shadcn <https://ui.shadcn.com/>

- Material-UI <https://mui.com/>