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
  - Needed for:
    - Add 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

State

$$f\left(\begin{array}{l} name: John \\ surname: Dough \end{array}\right) =$$

View

John

Dough

OK

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
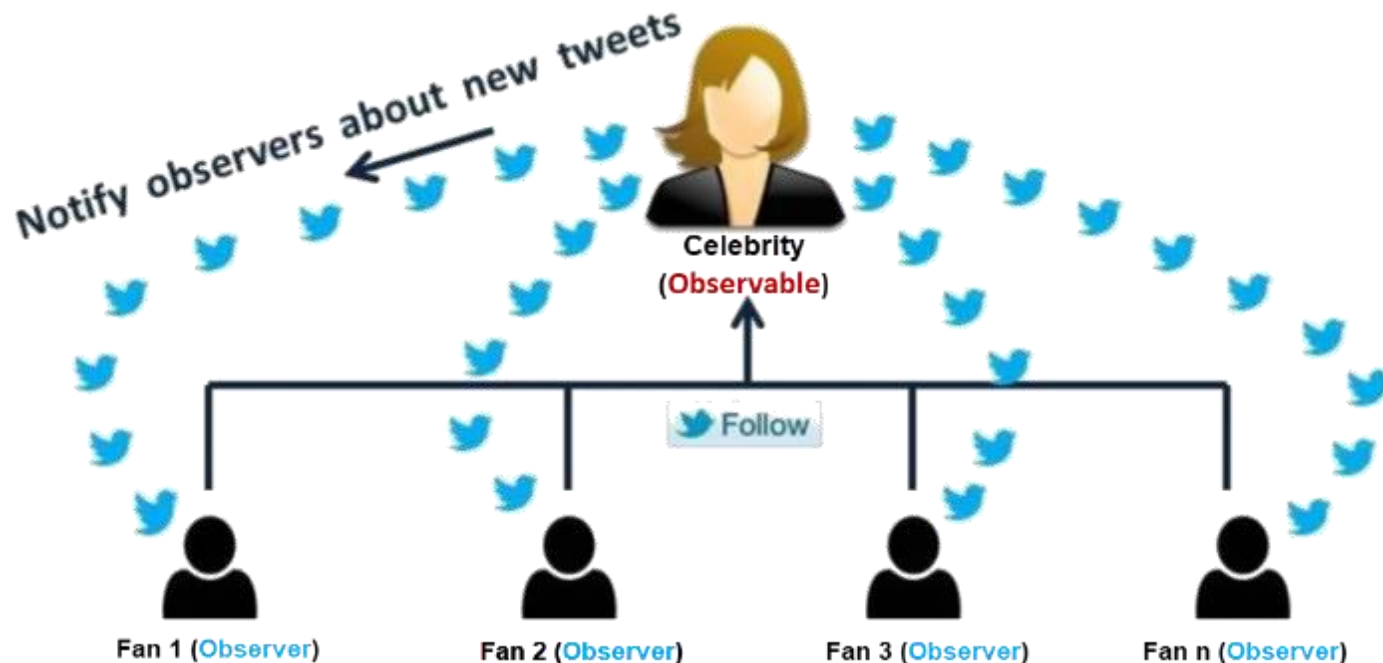
State

- 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

  - 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

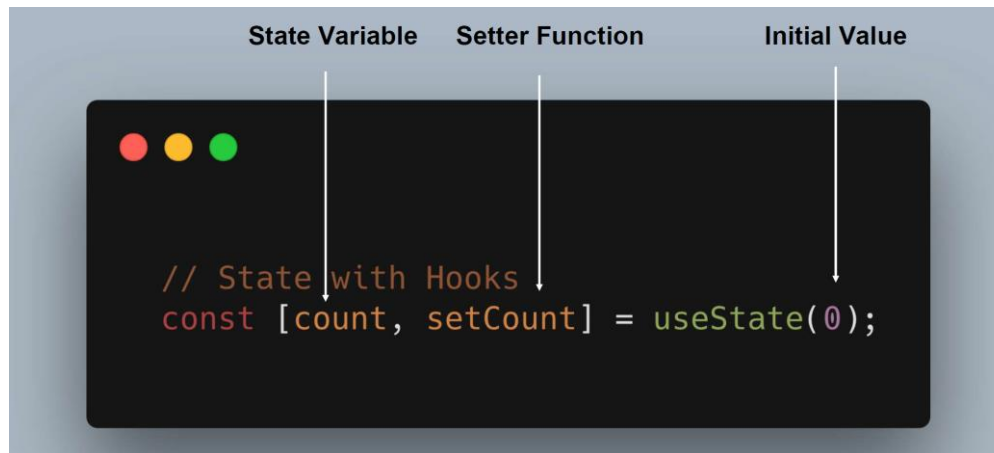**const [state, setState] = useState(initialState)**

The name of your state

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

The initial value of your state

State Variable   Setter Function   Initial Value

```
// State with Hooks
const [count, setCount] = useState(0);
```

Component with State + Events Handling

Count: 4 [+][-]

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

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;
```
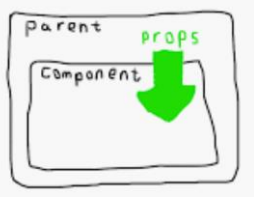
> **Handling events** is done the way events are handled on DOM elements

- Use the **Counter** component

```jsx
<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 and 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**
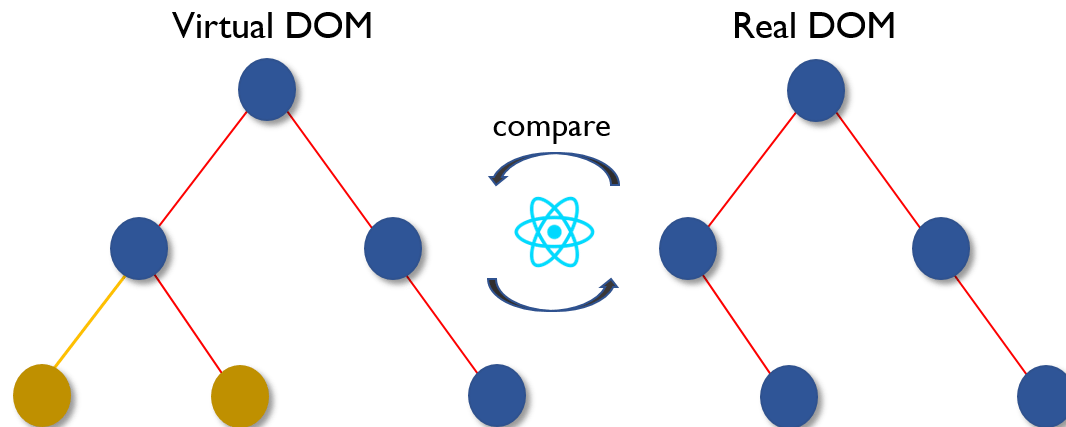
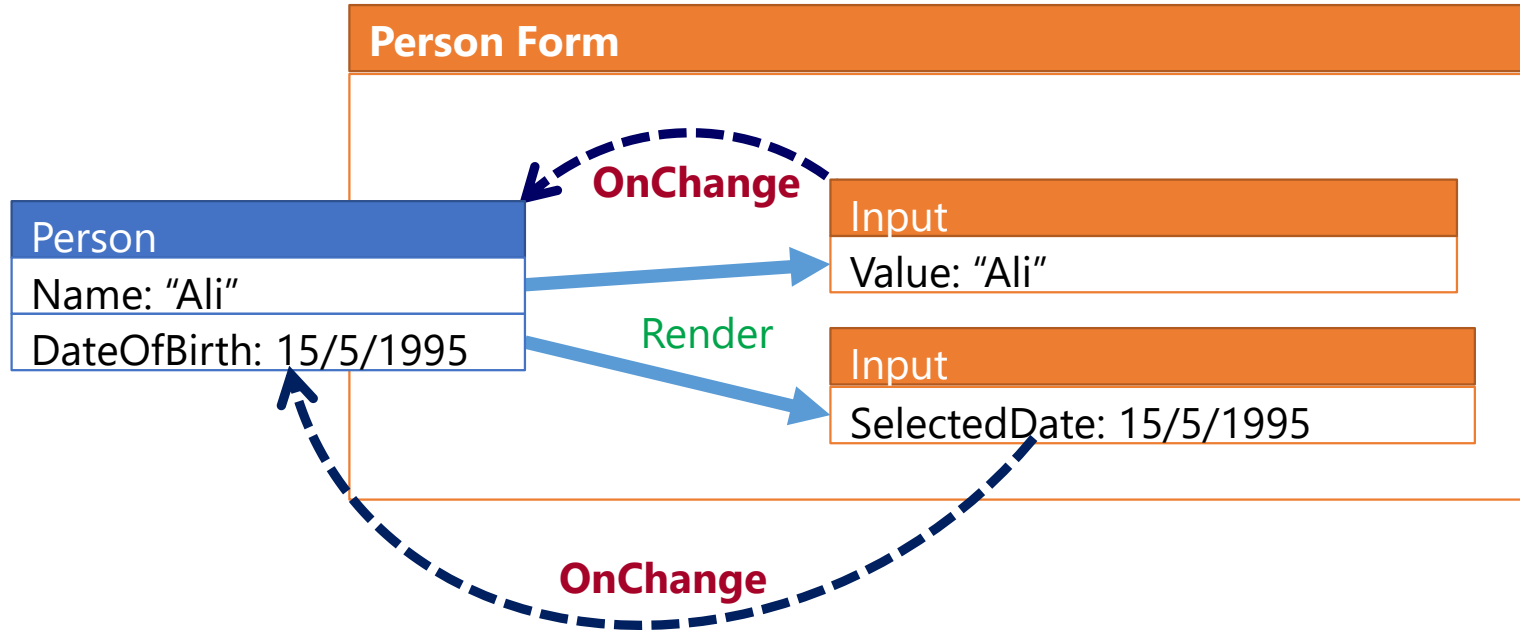# Event Handlers update the State and Reacts updates the UI
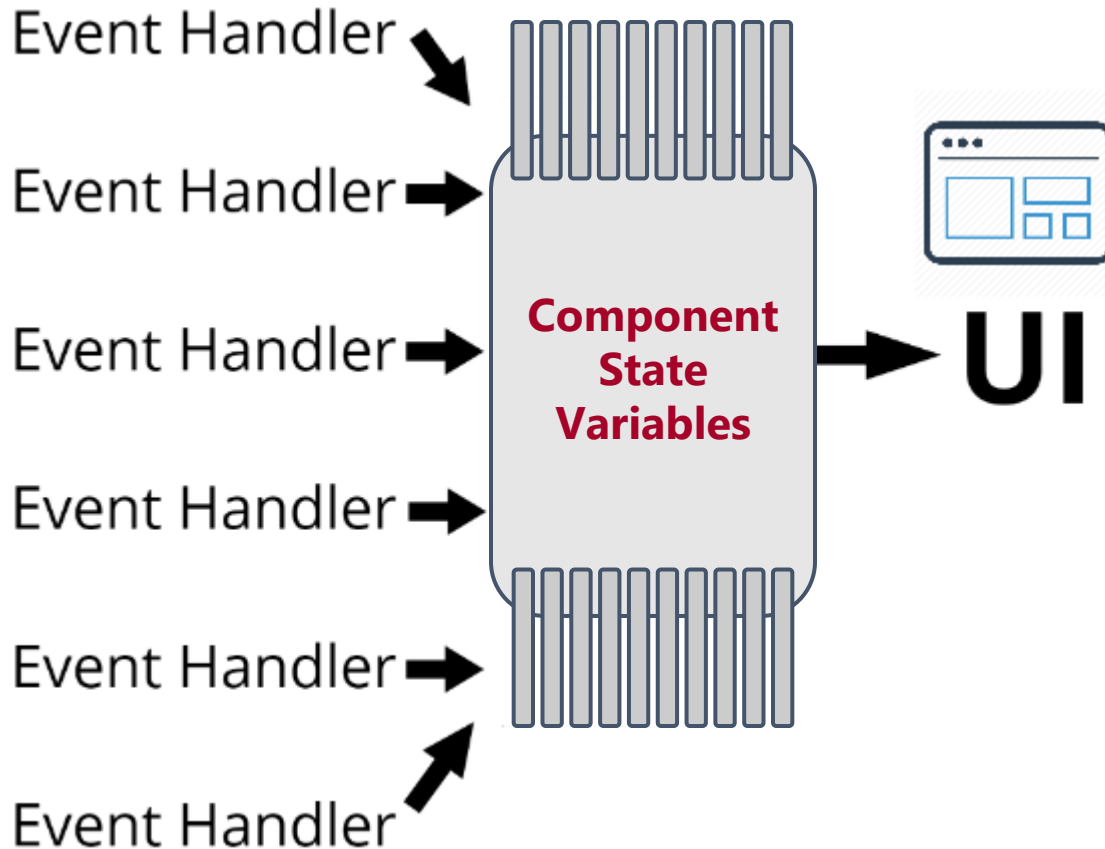
Every place a state variable is displayed is guaranteed to be auto-updated

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**

  o A new virtual DOM tree is generated

  o New tree is **diffed** against old…

  o …producing a minimum set of changes to be performed on real DOM to bring it up to date

Virtual DOM   compare   Real DOM

Unidirectional Data Flow in Forms

Person Form

Person
Name: "Ali"
DateOfBirth: 15/5/1995

OnChange

Input
Value: "Ali"

Render

Input
SelectedDate: 15/5/1995

OnChange

Common Events: onClick - onSubmit – onChange

Forms with React

```jsx
<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>
```

Form UI

```js
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));
};
```
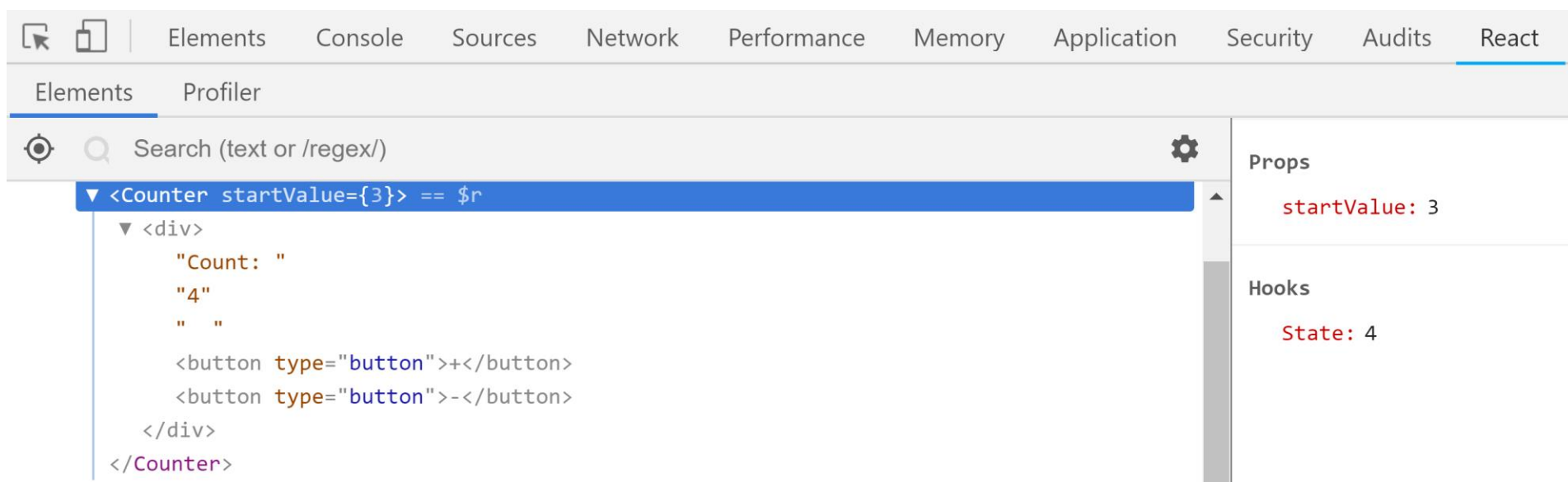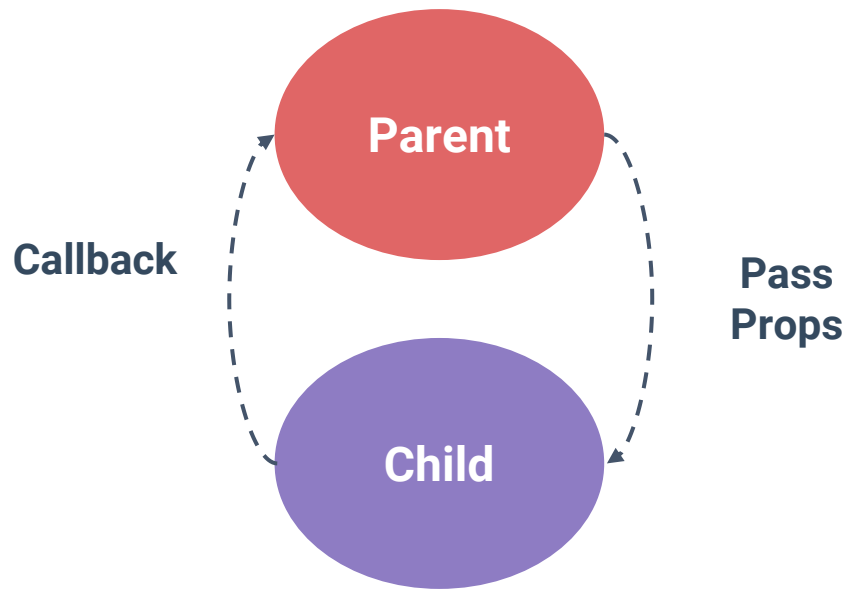
Form State and Event Handlers

React Dev Tools

- React Dev Tools
  https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en

Components Communication

Parent

Child

Callback

Pass Props
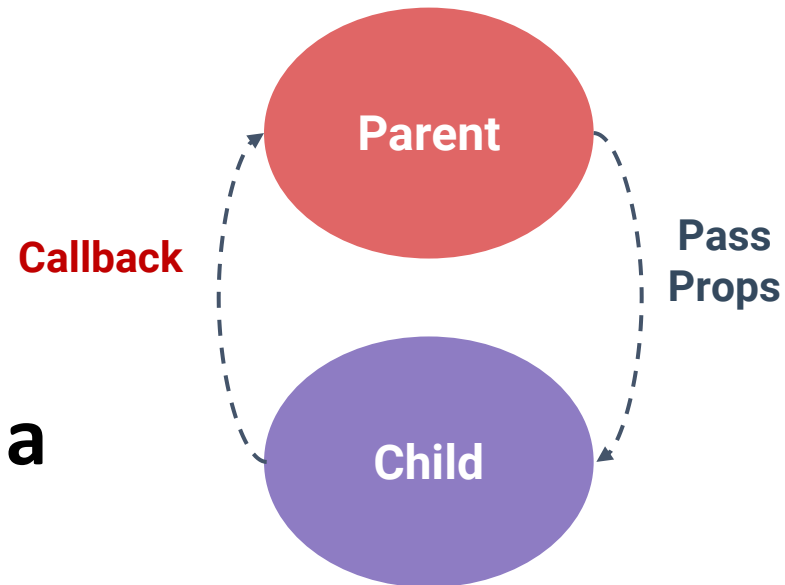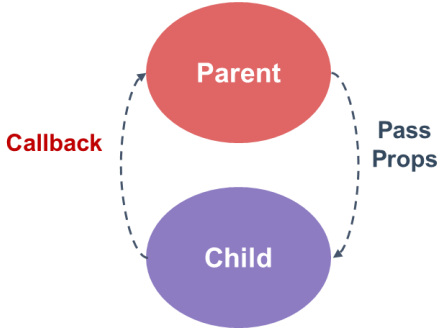
Composing Components

- 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

Callback

Pass Props

Child

Parent-Child Communication

Parent

Child

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


     function Counter(props) {
         const [count, setCount] = useState(props.startValue);

         const increment = () => {
             const updatedCount = count + 1;
             setCount(updatedCount);
             props.onChange(updatedCount);
         };

         return <div>
             Count: {count}
             <button type="button" onClick={increment}>+</button>
         </div>
     }
```

21

Common React Hooks

React Hooks

1. useState

2. useEffect

3. useRef

4. useContext

5. useRouter

6. useActionState

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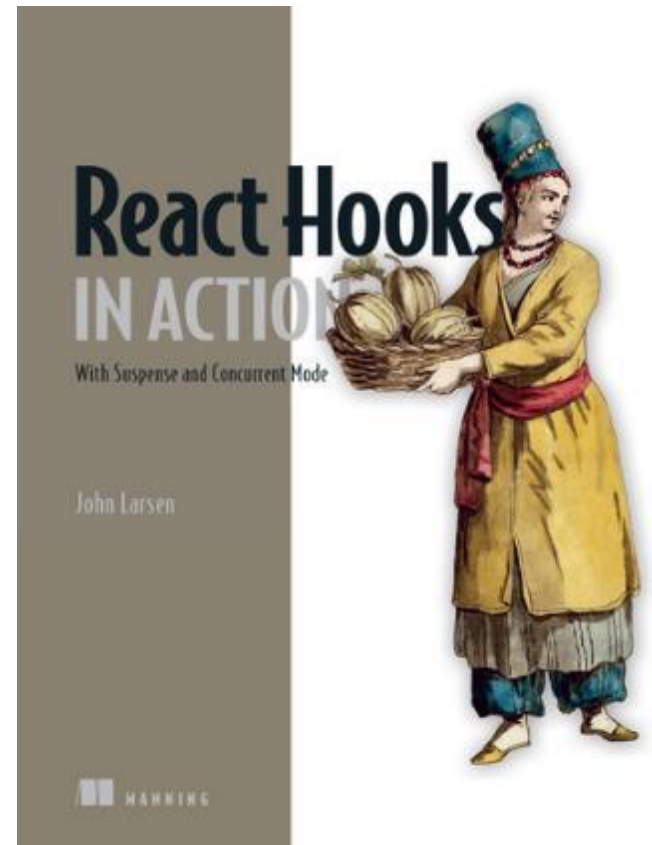
Some of the slides are based on
https://learning.oreilly.com/library/view/
react-hooks-in/9781617297632/

What is Hook?

- A Hook is a special function that lets you **hook** into React features such as state and lifecycle methods

- There are 3 rules for hooks:

  - Hooks can only be called at the top level of a component

  - Hooks cannot be conditional

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.).
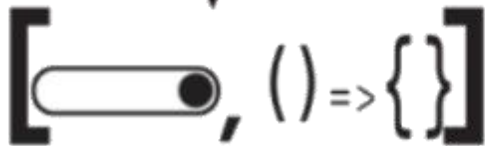
Dependency list:
Run the effect only if the values in the array change.
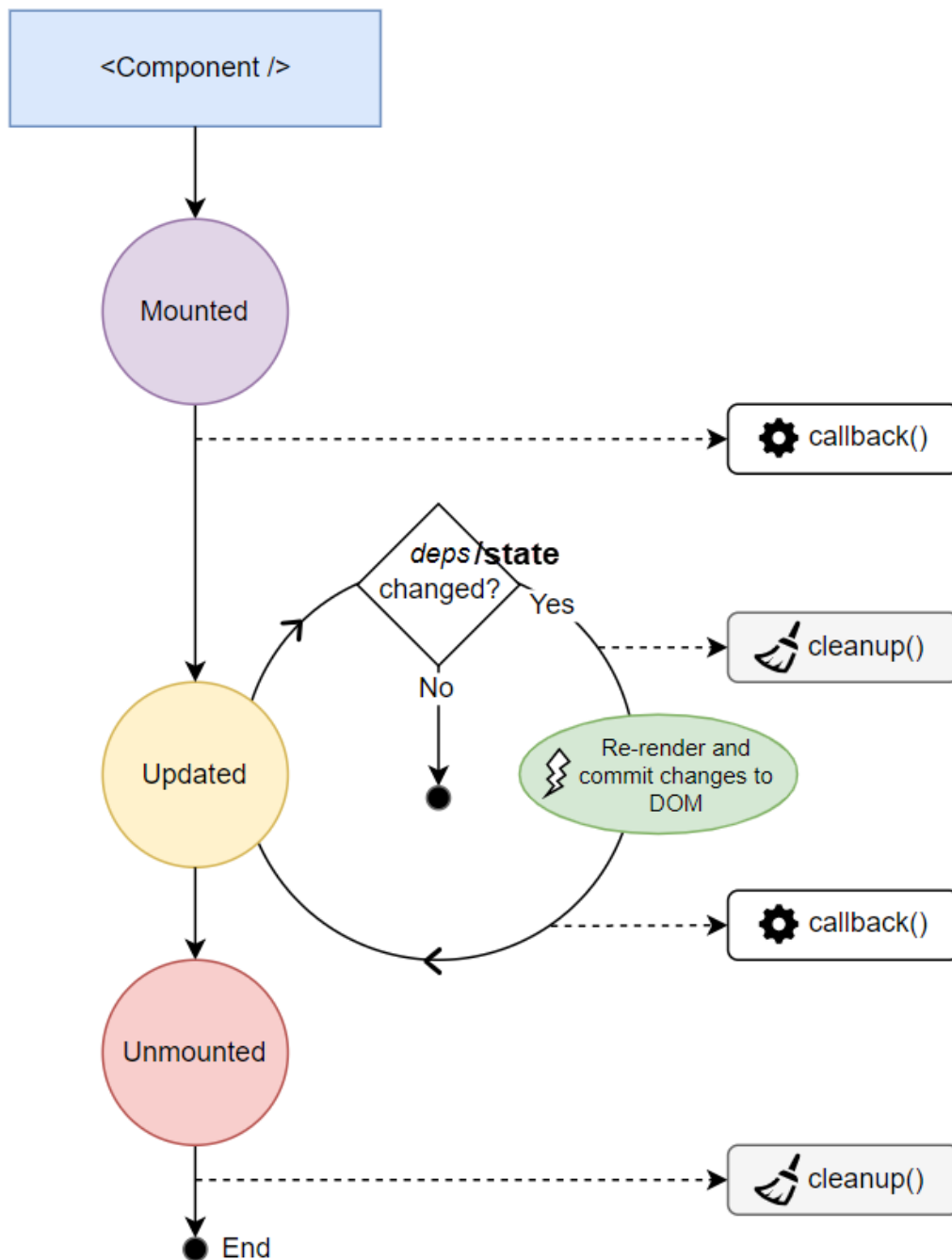
A) After initial rendering, `useEffect()` invokes the callback having the side-effect. cleanup function is not invoked

B) On later renderings, before invoking the next side-effect callback, `useEffect()` invokes the cleanup function from the previous side-effect execution (to clean up everything after the previous side-effect), then runs the current side-effect

C) Finally, after unmounting the component, `useEffect()` invokes the cleanup function from the latest side-effect

Common side effects

Common side effects include:

- Setting the page title imperatively

- Working with timers like `setInterval` or `setTimeout`

- Logging messages to the console or other service

- Fetching data or subscribing and unsubscribing to services

- Setting or getting values in local storage

useEffect - 2nd argument

[ ]	...nothing...	[data]
Run at initial render	Run at initial render	Run at initial render
	Run after every re-render	Run after every re-render if data has changed since last render
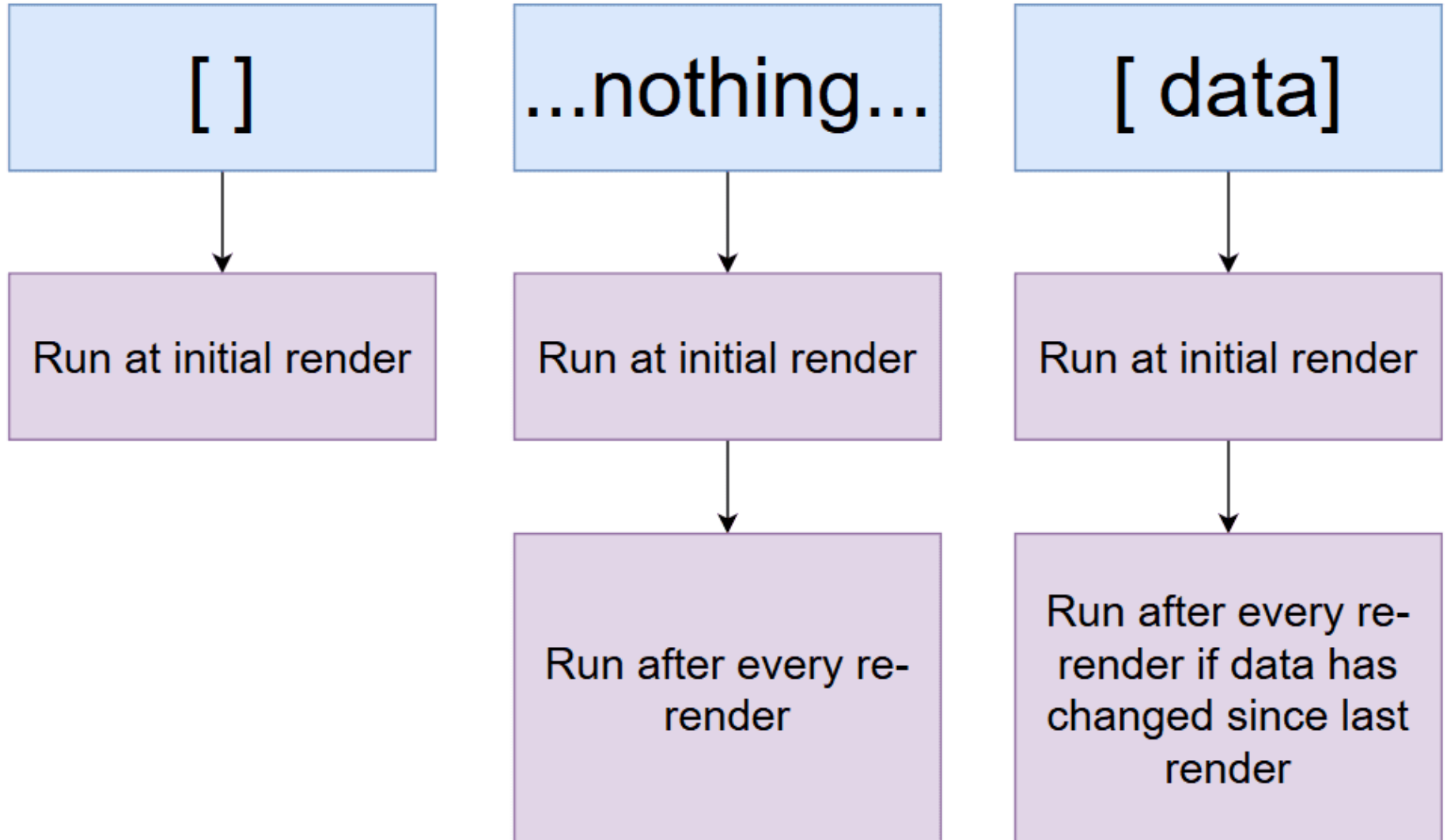
Use cases for the useEffect hook

Call pattern	Code pattern	Execution pattern

No second argument	```useEffect(() => {
	  // perform effect
	});```
		Run after every render.

Empty array as second argument	```useEffect(() => {
	  // perform effect
	}, []);```
		Run once, when the component mounts.

Dependency array as second argument	```useEffect(() => {
	  // perform effect
	  // that uses dep1 and dep2
	}, [dep1, dep2]);```
		Run whenever a value in the dependency array changes.

Return a function	```useEffect(() => {
	  // perform effect
	  return () => {/* clean-up */};
	}, [dep1, dep2]);```
		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)
  - o The value of the reference is persisted (stays the same) between component re-renderings
  - o Updating a reference doesn't trigger a component re-rendering.

Initial value:
Pass the initial value to the useRef hook.

```
const refObject = useRef( initialValue );
```

{ Current }

Ref:
React returns an object with a current property.

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

```javascript
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

```jsx
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**.
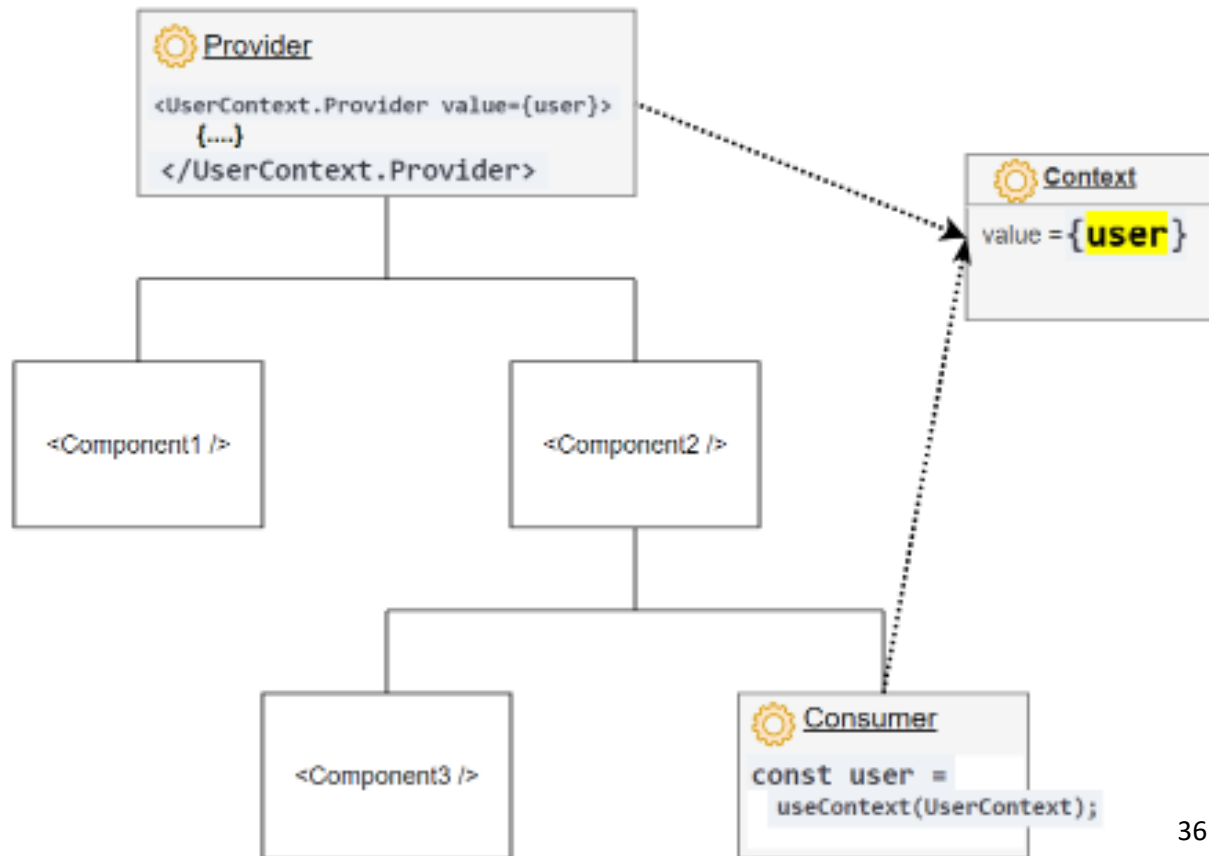
useRef vs. useState

- 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

  o 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

```
Provider
<UserContext.Provider value={user}>
    {....}
</UserContext.Provider>
```

```
Context
value = {user}
```

```
<Component1 />
```

```
<Component2 />
```

```
<Component3 />
```

```
Consumer
const user =
    useContext(UserContext);
```

useContext – provides shared variables and functions

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

```jsx
import React from 'react';
const UserContext = React.createContext();
export default UserContext;
```

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

```jsx
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**

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

Shared State Example

Shared state: App manages the shared user state for the app.

user ●

App

2. New context value: When App re-renders, it sets the latest state value on the provider.

1. Update state: When the state is updated by calling setUser, the App component re-renders.

`value={{user, setUser}}`

Provider

user ●

UserPicker

BookingsPage

3. Intermediate components: All the components down the tree re-render, even if they don't use context.

4. Context state: Components that consume the context access the latest value.

Bookings

BookingDetails

Context

user ●

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

```jsx
'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

- useActionState is a new React hook that manages the state of a form submission

  - It simplifies form logic by tracking pending, success, or error states after an action runs

  - Built-in support for server actions

  - Returns [state, dispatch, isPending]

  - Ideal for handling form submissions (e.g., login, register, feedback)

  - https://react.dev/reference/react/useActionState

- ToDo: More details soon…

useOptimistic

- `useOptimistic`: allows showing instant feedback to users before the server confirms a mutation (like adding a comment, updating a list, etc.)

  - It lets you optimistically update the UI to reflect the expected result of a server action, while still calling that action in the background

  - Think of it like: "Act as if the update already happened while we wait for the real response"

  - Keeps UI responsive: Improves user experience

```jsx
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>
  );
}
```

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

Common patterns for interleaving Server and Client Components

Pattern	Description	Use Case	Benefits
Server Wrapper, Client Core	Server fetches data and passes it to client component	Dashboard, data-driven UIs	Fast render, clear separation
Client Component with Server Actions	Client triggers server logic via form action or useActionState	Forms, to-dos, CRUD apps	Clean server logic, smaller client bundle
Server Component with Client Children	Server renders parent layout, client handles interactive children	Modals, filters, dynamic widgets	Reusable layout with local interactivity

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/