◇ What is Hosting?

Hosting refers to the service of **storing and serving websites, applications, or services on the internet**. When you create a website or an app, it needs to live on a server—a powerful computer that's always connected to the internet. Hosting providers give you space on these servers.

◇ Where to Host?

There are several **types of hosting platforms**, depending on your need:

Hosting Type	Ideal For	Examples
Shared Hosting	Beginners, small websites	Hostinger, Bluehost, GoDaddy
VPS Hosting	Medium apps/websites, more control	DigitalOcean, Linode, Kamatera
Cloud Hosting	Scalable apps, high traffic	AWS, Google Cloud, Microsoft Azure
Dedicated Hosting	Enterprise apps, full control	OVH, Liquid Web
Managed Hosting	WordPress or specific stack users	Kinsta, WP Engine
Static Hosting	Simple HTML/CSS sites	Netlify, Vercel, GitHub Pages

◇ Which Hosting to Choose?

It depends on **what you're building**:

☑ For Students / Portfolios / Static Sites:

- **GitHub Pages** (free)
- **Netlify** / **Vercel** (easy CI/CD, free tier)

☑ *For Full-stack Projects:*

- **Heroku** (simple deployment, limited free tier)
- **Render** (better than Heroku for free apps)
- **Railway.app** (great for beginners)
- **AWS EC2 + S3** (if you want cloud experience)

☑ *For Professional Cloud/DevOps Practice:*

- **AWS** (EC2, S3, RDS, etc.)
- **Google Cloud Platform**
- **Azure**

☑ *For Blogging / WordPress:*

- **Hostinger**, **Bluehost** (cheap, beginner-friendly)
- **Kinsta** (for premium managed WordPress)

HOOKS:

1. `useState` – Counter App :

Sample code:

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

function Counter() {
 const [count, setCount] = useState(0);
 return (
  <div style={{ textAlign: 'center', marginTop: '2rem' }}>
   <h1>Count: {count}</h1>
   <button onClick={() => setCount(count + 1)}>Increment</button>
  </div>
 );
}
export default Counter;
```

- `useState(0)` initializes a state variable `count` with the value `0`.
- `setCount` is the function used to update the `count`.
- Every time you click the button, `setCount(count + 1)` updates the value, and the component re-renders.

2. 2. `useEffect` – Fetching Data:

```
import { useState, useEffect } from "react";
function Home() {
 const [count, setCount] = useState(0);
 const [data, setData] = useState('');
 const handleChange = (e) => {
  setData(e.target.value);
 }
 useEffect(() => {
  setCount((count) => count + 1);
 }, [data]);
 return (
  <>
   <input onChange={handleChange} value={data} />
   <p>{count}</p>
  </>
 );
}

export default Home;
```

- `useEffect` runs **once after the component mounts** ([ ] means no dependencies).
- We fetch users from an API and update the `users` state with `setUsers`.
- It simulates a **componentDidMount** lifecycle method.

`useEffect` is used for side effects like data fetching, setting up listeners, or manipulating the DOM.

3. `useContext` – Theme Example :

```
import React, { useContext, useState, createContext } from 'react';

// 1. Create a Context
const ThemeContext = createContext();

function ThemeToggleButton() {
 // 2. Use the context value
 const { theme, toggleTheme } = useContext(ThemeContext);
```

```
  return (
    <button onClick={toggleTheme}>
      Switch to {theme === 'light' ? 'Dark' : 'Light'} Mode
    </button>
  );
}

function ThemedContent() {
  const { theme } = useContext(ThemeContext);
  const style = {
    padding: '20px',
    background: theme === 'light' ? '#f9f9f9' : '#333',
    color: theme === 'light' ? '#000' : '#fff',
  };

  return <div style={style}>This is {theme} mode content.</div>;
}

function App() {
  // 3. Manage context state in a parent component
  const [theme, setTheme] = useState('light');

  const toggleTheme = () =>
    setTheme((prevTheme) => (prevTheme === 'light' ? 'dark' : 'light'));

  return (
    // 4. Provide the context value to children
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      <div style={{ textAlign: 'center', marginTop: '2rem' }}>
        <h2>useContext Theme Example</h2>
        <ThemeToggleButton />
        <ThemedContent />
      </div>
    </ThemeContext.Provider>
  );
}

export default App;
```

. `createContext()` creates a context to share data.

. `useContext(ThemeContext)` lets child components consume that data.

. The App component manages the `theme` state and passes it down.

. `ThemeContext.Provider` makes the value available to all children inside it.

4, `useRef` – Focusing Input :

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


function App() {
  const inputRef = useRef(null); // Create a reference to the input element


  const handleFocus = () => {
    inputRef.current.focus(); // Focus the input when button is clicked
  };


  return (
    <div style={{ textAlign: 'center', marginTop: '50px' }}>
      <h2>useRef Example – Focus Input</h2>
      <input
        ref={inputRef}
        type="text"
        placeholder="Click the button to focus me"
```

```
      style={{ padding: '10px', fontSize: '16px' }}
    />
    <br /><br />
    <button onClick={handleFocus} style={{ padding: '10px 20px' }}>
      Focus Input
    </button>
  </div>
 );
}

export default App;
```

1. `useRef(null)`	Creates a reference object (`inputRef`) with `.current = null` initially
2. `ref={inputRef}`	Attaches that ref to the `<input>` element
3. `inputRef.current.focus()`	Directly accesses the DOM element and calls its `focus()` method

5. `useReducer` – Counter with Reducer:

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

// Step 1: Define the reducer function
function reducer(state, action) {
```

```
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'reset':
      return { count: 0 };
    default:
      return state;
  }
}

function App() {
  // Step 2: useReducer returns [state, dispatch]
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <div style={{ textAlign: 'center', marginTop: '50px' }}>
      <h2>useReducer Counter Example</h2>
      <h1>{state.count}</h1>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
      <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
```

```
    <button onClick={() => dispatch({ type: 'increment' })}>+</button>
   </div>
  );
}


export default App;
```

`1. reducer()`	A function that takes current state + action and returns new state
`2. useReducer(reducer, { count: 0 })`	Initializes state with `{ count: 0 }` and provides `dispatch()`
`3. dispatch({ type: 'increment' })`	Sends an action to the reducer to update the state

6. `useCallback` – Preventing Unnecessary Renders :

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

// Child component

const Child = React.memo(({ onClick }) => {

  console.log("Child rendered");

  return <button onClick={onClick}>Click Me (Child)</button>;

});

// Parent component

function App() {

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

  const handleClick = useCallback(() => {
```

```
  console.log("Button clicked");
 }, []);

 return (
  <div>
   <h2>Count: {count}</h2>
   <button onClick={() => setCount(count + 1)}>Increment Count</button>
   <Child onClick={handleClick} />
  </div>
 );
}
export default App;
```

7. `useMemo` – Expensive Calculation:

```
import React, { useState, useMemo } from 'react';

function App() {
 const [number, setNumber] = useState(1);
 const [darkMode, setDarkMode] = useState(false);

 // Step 1: Expensive factorial calculation function
 function expensiveFactorial(n) {
  console.log('Calculating factorial...');
```

```
  let result = 1;
  for (let i = 1; i <= n; i++) {
    // Simulate heavy computation
    for (let j = 0; j < 100000000; j++) {}
    result *= i;
  }
  return result;
}

// Step 2: useMemo to cache the result
const factorial = useMemo(() => expensiveFactorial(number), [number]);

const themeStyles = {
  backgroundColor: darkMode ? '#333' : '#fff',
  color: darkMode ? '#fff' : '#000',
  padding: '20px',
  textAlign: 'center',
};

return (
  <div style={themeStyles}>
    <h2>useMemo – Expensive Calculation</h2>
```

```
    <input
      type="number"
      value={number}
      onChange={(e) => setNumber(parseInt(e.target.value) || 1)}
      min="1"
      style={{ padding: '10px', margin: '10px' }}
    />
    <br />
    <button onClick={() => setDarkMode(!darkMode)}>
      Toggle {darkMode ? 'Light' : 'Dark'} Mode
    </button>
    <h3>Factorial of {number}: {factorial}</h3>
   </div>
  );
}


export default App;
```

1. `expensiveFactorial(number)`	Simulates a heavy CPU task (factorial with delay)	
2. `useMemo(() => expensiveFactorial(number), [number])`		Only recalculates factorial **if `number` changes**
3. `darkMode` toggle	Lets you test that the factorial **doesn't recalculate unnecessarily** when other states change	