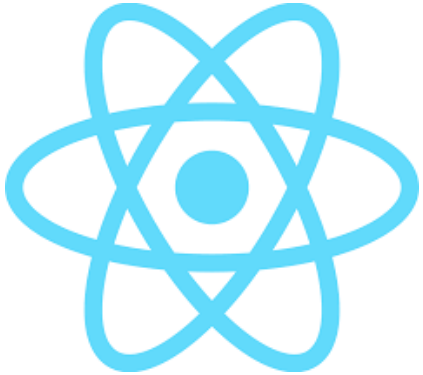
REACT

ONE LOVE. ONE FUTURE.

Motivation

- Understand the core philosophy of React and how it powers modern Single Page Applications
- Grasp how React achieves fast, seamless user experiences through Virtual DOM and component-based design.
- Learn to build scalable, reusable, and maintainable web interfaces using React's declarative approach.
- Prepare to advance toward modern frontend frameworks (Next.js, React Native) and advanced topics (Redux, Next.js, Server Components, etc.)



Content

1. Single Page Application
2. React Core Concepts
3. React Hooks
4. Routing and Context API

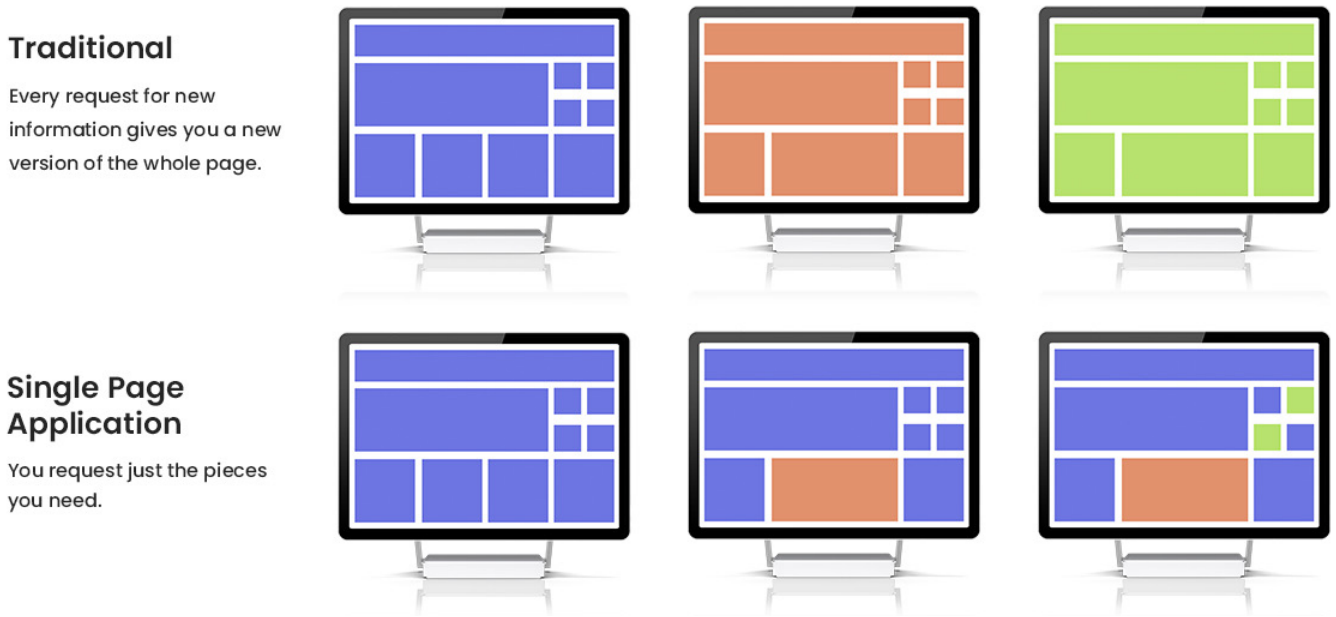
HUST

hust.edu.vn fb.com/dhbkhn

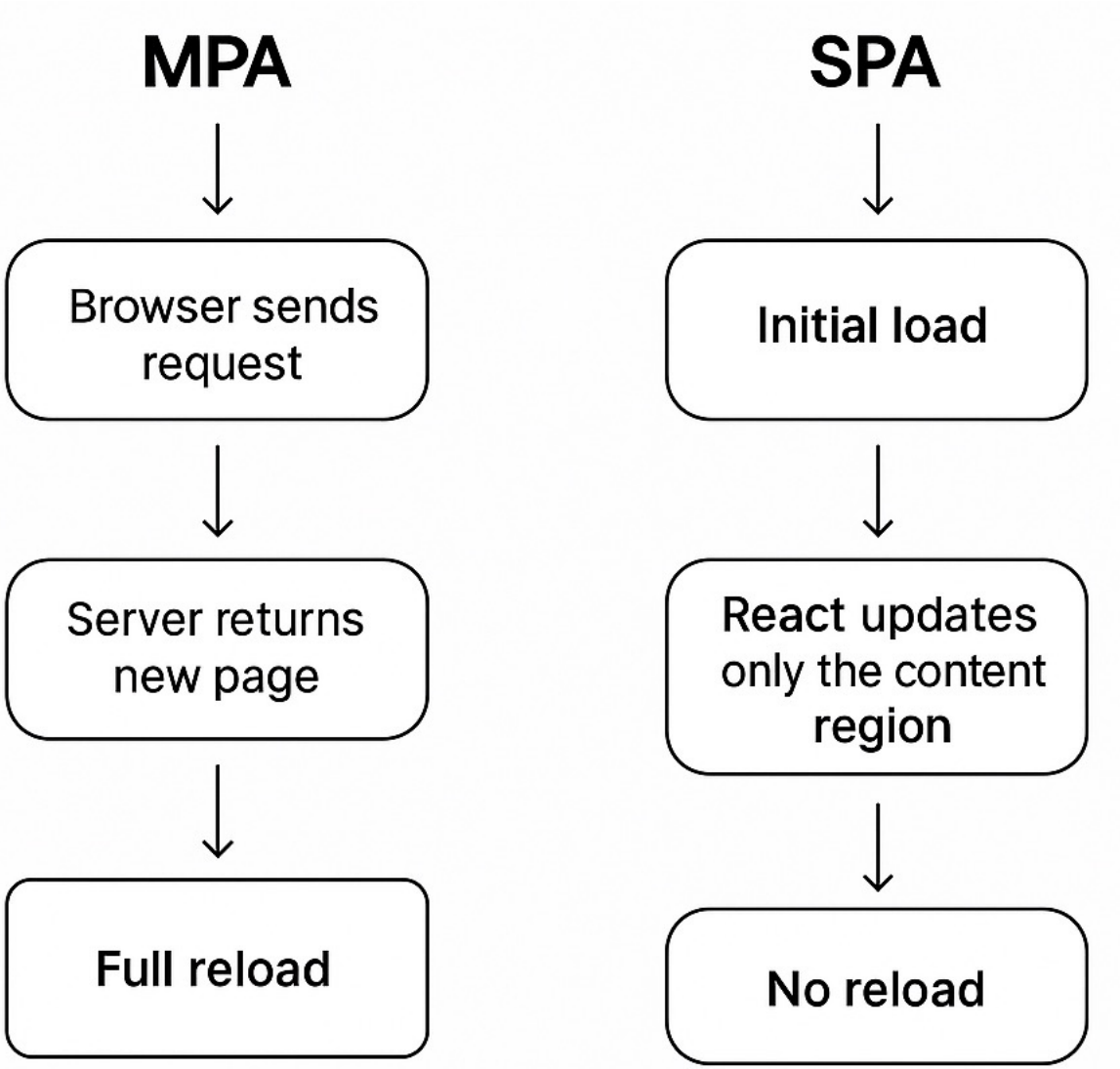
1. Single Page Application

Multi Page Application vs Single Page Application

- Traditional web applications perform most of the application logic on the server
- Single Page applications (SPAs) perform most of the user interface logic in a web browser, communicating with the web server primarily using web API



Multi Page Application vs Single Page Application



What is a Single Page Application?

A Multiple Page App

- Reloads the entire page on each request
- Every change requests rendering a new page from the server => browser reloads the content of a page completely and downloads the resources again

A Single Page App

- Loads a single HTML page and dynamically updates its content
- Uses JavaScript (usually React, Angular, or Vue) to update parts of the interface
- Examples: Gmail, Facebook, Tiki.

Multi Page Application vs Single Page Application

Criteria	Multi-Page Application	Single-Page Application
Advantages	• Better SEO (each page has a unique URL) • Suitable for large content-heavy sites • Easier server-side security and access control	• Fast and smooth user experience • Less server load after initial request • Great for dynamic and interactive UIs
Disadvantages	• Slower navigation (full page reload each time) • Higher bandwidth and server usage • Poorer user experience for frequent transitions	• Harder SEO optimization • Large initial load • Relies heavily on client-side JavaScript

React Development Environment: JSX, Vite

- JSX (JavaScript XML)
  - JSX is a syntax extension for JavaScript used in React.
  - You write HTML-like code inside JavaScript
  - JSX is transpiled into regular JavaScript functions
  - *const element = <h1>Hello, React!</h1>;*

- Vite –Modern React Development Tool
  - Vite is a fast frontend build tool for frameworks/libs like React.
  - Uses ES Modules and esbuild for extremely fast compilation.
  - Commonly replaces *Create React App (CRA)* for new React projects.

HUST

hust.edu.vn fb.com/dhbkhn

2. React Core Concepts

2.1. Virtual DOM

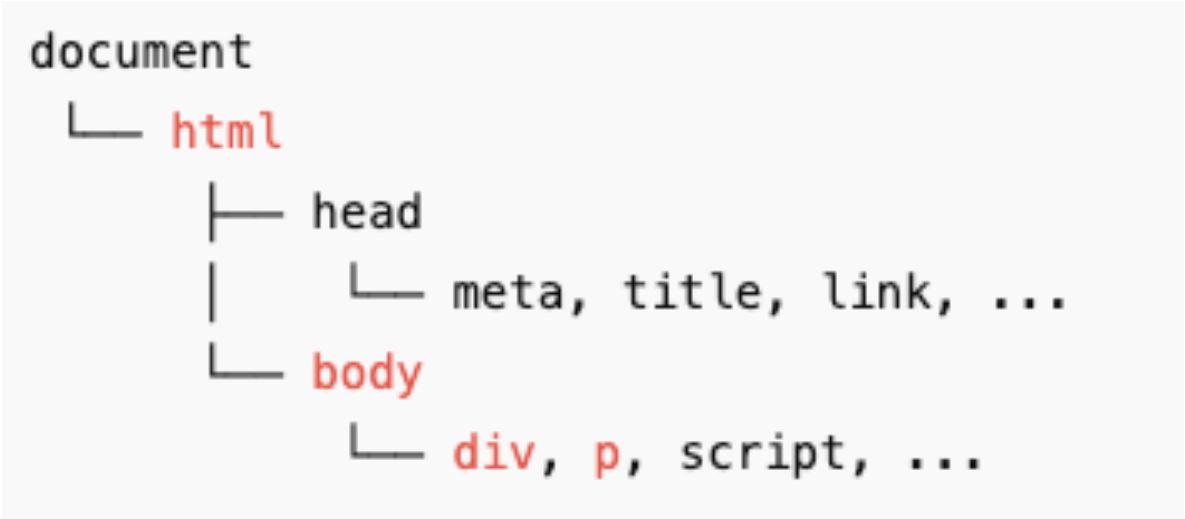
2.2. Component

2.3. Props

2.4. State

Traditional DOM (Real DOM)

- DOM (Document Object Model) is a tree-like representation of an HTML document.
- Every element (<html>, <body>, , <p>, etc.) is a node in the tree.
- The browser parses HTML and builds this DOM tree so JavaScript can access and modify page elements.



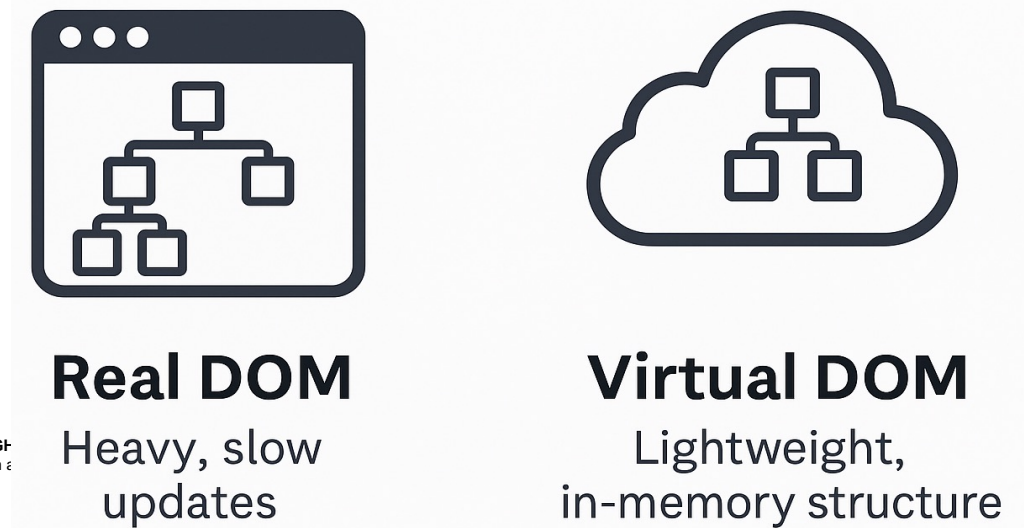
Traditional DOM (Real DOM)

Limitations of the Real DOM

- Poor performance: when a change happens, the browser must recalculate styles, reflow layout, and repaint interface the entire page or large parts of it.
- Inefficient updates: real DOM is not optimized for applications with frequent updates (like interactive dashboards or live data).
- Complex management: browser cannot compare states (nodes and data) — it just rebuilds as needed, leading to performance issues.

Virtual DOM

- It's a JavaScript object, in-memory representation of the Real DOM
- Key concept
  - "In-memory": Virtual DOM is stored and updated inside RAM, not directly in the browser's document structure.
  - React can compute updates efficiently before applying them



Virtual DOM in React

- When data changes, React performs three key steps
- Step 1: Render
  - React renders a new Virtual DOM tree based on the updated data.
- Step 2: Diff
  - React uses diffing algorithm to compare the new Virtual DOM with the previous version
  - React determines the minimal set of changes required to update the Real DOM.
- Step 3: Patch
  - React applies patches (detected differences) to the Real DOM.
  - Only the changed parts of the DOM are updated — everything else remains untouched.

React Diffing Algorithm (Reconciliation)

- React uses simple rules to checking:
  - Different type of elements (e.g., to): different subtrees (destroy & rebuild)
  - Same type of elements: compare properties (e.g., className).
  - Child list: React uses key property to identify elements
    - created by developer
    - index in the list
- React Fiber
  - Reconciliation engine introduced in React 16
  - Change: synchronous update -> asynchronous, interruptible, and prioritized.

```
<ul>
  <li key="1">Apple</li>
  <li key="2">Banana</li>
</ul>
```

```
<ul>
  <li key="1">Orange</li>
  <li key="2">Banana</li>
</ul>
```

Virtual DOM - Performance Benefits

- React's Virtual DOM reduces unnecessary re-renders and speeds up UI updates
  - Detects which elements have changed
  - Applies minimal updates to the Real DOM.

Operation	Real DOM	Virtual DOM
Full-page re-render	❌ Slow (entire layout recalculated)	✅ Fast (only changed nodes updated)
Frequent updates	Causes lag and jank	Smooth and optimized
Ideal for	Static content	Dynamic, interactive UIs
Manipulated by	Browser	React

Component

- Component is a reusable, independent building block that defines part of the user interface (UI).
  - Template (UI): How it looks (HTML/JSX).
  - Logic: How it behaves (JavaScript functions, state, events).
  - Style: How it's styled (CSS or inline styles).
- Use Components like HTML tags

```jsx
function Welcome() {
  return (
    <h1 style={{ color: "#1976d2", textAlign: "center" }}>
      Hello, React!
    </h1>
  );
}
```

jsx

```jsx
<Welcome />
```

Component-Based Architecture

- React applications are built using the Components
  - UI is divided into small, independent, and reusable pieces, called components.
  - Each component handles its own logic, layout, and style, and can be combined to form complex interfaces.

```
<App>
  ├── <Header />
  ├── <Sidebar />
  ├── <MainContent>
  │     └── <CardList>
  │           └── <CardItem>
  │                 └── <Button />
  └── <Footer />
```

Functional vs Class Components

- Two main ways to create Components:
  - Class Components: traditional, object-oriented approach (before React 16.8).
  - Functional Components: modern, simpler approach

Aspect	Class Component	Functional Component
Declaration	class keyword	function keyword
State	this.state, setState()	useState() Hook
Lifecycle	Class methods (componentDidMount, etc.)	useEffect() Hook
Readability	Longer	Shorter, cleaner

Functional vs Class Components

```jsx
class Counter extends React.Component {
  state = { count: 0 };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          +1
        </button>
      </div>
    );
  }
}
```

Class Component

```jsx
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

Functional Component

+1

Component Tree: Parent–Child Interaction

- UI is composed of multiple components arranged in a Component tree.

Direction	Mechanism	Description
🔽 Parent → Child	Props	Parent passes data to child.
🔼 Child → Parent	Callback Function (event)	Child calls a function to send data to parent
🔁 Sibling ↔ Sibling	Parent or Context	Through their common parent (or Context API).

Component Tree: Parent–Child Interaction

```jsx
// Child component
function Child({ message, onNotify }) {
  return (
    <div>
      <p>Child received: {message}</p>
      <button onClick={() => onNotify("Hello from Child!")}>
        Send message to Parent
      </button>
    </div>
  );
}


// Parent component
function Parent() {
  const [childMsg, setChildMsg] = React.useState("");

  // Callback function for the child
  const handleChildMessage = (msg) => setChildMsg(msg);

  return (
    <div>
      <h2>Parent Component</h2>
      <p>Message from Child: {childMsg}</p>

      {/* Passing data (props) and callback */}
      <Child message="Hi from Parent!" onNotify={handleChildMessage} />
    </div>
  );
}
```

Message from Child:

Child received: Hi from Parent!

Send message to Parent

Message from Child: Hello from Child!

Child received: Hi from Parent!

Send message to Parent

Props

- Props (properties) are arguments passed from a parent component to a child component
- Props are read-only — a child component cannot modify the props it receives.

```
function UserCard(props) {
  return <h3>Hello, {props.name}!</h3>;
}

function App() {
  return (
    <div>
      <UserCard name="Lam" />
      <UserCard name="Minh" />
    </div>
  );
}
```

Component	Role	Data Flow
<App />	Parent	Sends data via props
<UserCard />	Child	Receives props and displays them
name="Lam"	Prop value	Passed from parent to child

Props are Read-only

- Props are like function parameters → the child can use them
- Props are immutable → they cannot be changed by the child component
- If you try to directly mutate props, it can lead to unexpected behavior and bugs unexpectedly.

```
function UserCard(props) {
  // ❌ Wrong: trying to modify props
  props.name = "Changed!";
  return <h3>Hello, {props.name}</h3>;
}

function App() {
  return <UserCard name="Lam" />;
}
```

Props - property name

- Child component can access data passed from the parent using the props object.
- Common pattern: props.propertyName

  → e.g., props.name, props.title, props.color.

```
function UserCard(props) {
  return <h3 style={{ color: props.color }}>Hello, {props.name}!</h3>;
}

function App() {
  return (
    <div>
      <UserCard name="Lam" color="blue" />
      <UserCard name="Mạnh" color="green" />
      <UserCard name="Thu" color="purple" />
    </div>
  );
}
```

Hello, Lam!

Hello, Mạnh!

Hello, Thu!

Props - children

- props.children represents any content nested inside a component's opening and closing tags.
- It allows components to wrap or compose other components.

```
function Card(props) {
  return (
    <div
      style={{
        border: "2px solid red",
        display: "inline-block",
      }}
    >
      {props.children}
    </div>
  );
}
```

```
function App() {
  return (
    <div>
      <Card>
        <Button />
      </Card>
    </div>
  );
}
```

{props.children} → replaced by <Button />,

```
function Button() {
  return <button>Click Me</button>;
}
```

Props - One way data flow

- Data flows in one direction: from parent to child
- Data always flows downward — from parent to child.
- Child cannot change the parent's data

Component	Role	Data Flow
Parent	Owns the data (state or variables)	Sends data down as props
Child	Receives data via props	Displays the data
Direction	Parent → Child	One-way flow only

State

- State is a component's internal data that React keeps track of.
- It determines how the UI looks at a given moment.
- When state changes, React automatically re-renders the component.
- State is mutable (can change), but must be updated through setState() or setCount() (in Hooks).

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

✅ count is a **state variable**
✅ setCount() updates it
➡️ Each update triggers a **UI re-render**

State vs Props

Aspect	Props	State
Origin	Received from parent component	Managed inside the component
Mutability	❌ Immutable – cannot be changed by the child	✅ Mutable – can be changed using setState() / setCount()
Purpose	Used to pass data and configure components	Used to store internal data that affects rendering

```
// Props example
<UserCard name="Lam" />

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

Props = external, fixed input data
State = internal, changeable data

useState Hook – Basic Example

- useState - a React Hook returns an array with two elements:
  - current state value
  - function to update it
- *const [state, setState] = useState(initialValue);*

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

useState Hook – Basic Example

```jsx
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

✅

```jsx
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => count++}>+1</button>
    </div>
  );
}
```

❌

State – Setter Function & Batching

- State updates are done through setter function (e.g. setCount).
- React does not update state immediately
- React combines multiple state updates to improve performance (batching)

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

  function handleClick() {
    console.log("Before:", count);
    setCount(count + 1);
    console.log("After:", count);
  }

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>+1</button>
    </div>
  );
}
```

```
Before: 0
After: 0
[Render] count = 1
```

Component Lifecycle

- Lifecycle: initial render => updates (re-renders) => unmounting.
- When state or props change, React re-renders that component
- React uses the Virtual DOM to compare changes and update the Real DOM efficiently.

```jsx
function Counter() {
  const [count, setCount] = useState(0);
  console.log("Render:", count);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

```
Initial Render:
 └ App
     └ Counter (Count = 0)
           ├ <p>Count: 0</p>
           └ <button>+1</button>

State changes → Re-render:
 └ App
     └ Counter (Count = 1)
           ├ <p>Count: 1</p>    ← updated node
           └ <button>+1</button>
```

HUST

3. Hook

3.1. useState

3.2. useEffect

Hook - Introduction

- Before React 16.8, developers used Class Components to manage: State, Lifecycle methods, Shared logic
- Problems
  - Complex lifecycle: logic in multiple methods
  - Long syntax: this, binding, constructor, etc.
- Hook: React 16.8 (2019)
  - ✅ Simplify component logic
  - ✅ Allow stateful logic inside function components
  - ✅ Replace many class lifecycle methods
  - ✅ Reuse logic easily through Custom Hooks

Hook - Introduction

- Hooks are **functions** that let you *"hook into"* React features (state, lifecycle, context, etc.) from **functional components**.
- Hooks = **Simpler lifecycle + State + Logic reuse**

```
Functional Component
        ↓
  useState() → Local state
        ↓
  useEffect() → Side effects (Lifecycle)
        ↓
  useContext() → Shared data (Global state)
        ↓
  React reconciles → Re-render with updated values
```

useState Hook

- useState - a React Hook returns an array with two elements:
  - current state value
  - function to update it
- *const [state, setState] = useState(initialValue);*

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

useState - Flow

```
function Demo() {
  console.log("Component function executed");

  const [count, setCount] = useState(0);
  console.log("useState called, count =", count);

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

```
Component function executed
useState called, count = 0
Component function executed
useState called, count = 1
Component function executed
useState called, count = 2
```

- React calls useState() on EVERY render
- useState() is called many times but its initial value is evaluated only during the first render.

Stage	Function / Process	Description
Initial Mount	Demo()	The component function is executed for the first time.
	useState(0)	React initializes a new state cell with the initial value 0.
	setCount	React creates and stores the updater function for this state.
Event Click Button	setCount(count + 1)	React queues a state update request — but doesn't update count immediately.
After Event	React's Scheduler	React detects a pending update and schedules a re-render for the component.
Re-render Begins	Demo() (again)	The entire component function is called again from the top.
	useState()	React retrieves the stored state (now 1), not the initial value.
Virtual DOM Diffing	React Reconciliation	React compares the new Virtual DOM tree with the previous one.
Real DOM Update	Browser Rendering	React patches only the changed parts (e.g., Count: 0 → Count: 1).

Lazy Initial State

- Compute only once when the initial value is expensive to calculate. useState(() => computeInitialValue())

```
function Example() {
  console.log("Component re-rendered");

  const [normal, setNormal] = React.useState(expensiveComputation());
  const [lazy, setLazy] = React.useState(() => expensiveComputation());

  function expensiveComputation() {
    console.log("expensiveComputation executed!");
    return 100;
  }

  return (
    <div>
      <button onClick={() => setNormal(n => n + 1)}>Normal: {normal}</button>
      <button onClick={() => setLazy(l => l + 1)}>Lazy: {lazy}</button>
    </div>
  );
}
```

```
Component re-rendered
expensiveComputation executed!
Component re-rendered
expensiveComputation executed!
```

```
Component re-rendered
expensiveComputation executed!
Component re-rendered
```

Updating State Based on Previous Value

- setCount(count + 1) may not always work as expected
- setCount() is asynchronous and batched: React may delay updates and merge them together.
- Solution: setState(prev => ...)

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

  function handleDoubleIncrease() {
    setCount(prev => prev + 1);
    setCount(prev => prev + 1);
  }
```

✅

```
function handleDoubleIncrease() {
  setCount(count + 1);
  setCount(count + 1);
}
```

❌

✅ **Result after one click:**
Count increases by **2**, not 1.

🚫 **Count** only increases by **1**

State as Object/Array – Keep It Immutable

- Immutability
  - State should be treated as **read-only**.
  - ❌ Don’t change the state value directly.
  - ✅ Always create a **new copy** and update that copy.

```
const [numbers, setNumbers] = useState([1, 2, 3]);
// ❌ Mutating state directly
function mutateAdd() {
  numbers.push(4);
  setNumbers(numbers);
  console.log("Mutated array:", numbers);
}

// ✅ Immutable update using spread operator
function immutableAdd() {
  const newNumbers = [...numbers, 4];
  setNumbers(newNumbers);
  console.log("Immutable array:", newNumbers);
}
```

State - Input Form

```jsx
function FormExample() {
  const [name, setName] = useState("");

  function handleChange(event) {
    setName(event.target.value);
  }

  return (
    <div>
      <h3>React Controlled Input</h3>
      <input
        type="text"
        placeholder="Enter your name"
        value={name}
        onChange={handleChange}
      />
      <p>Hello, {name || "guest"}!</p>
    </div>
  );
}
```

```
User typing "Lam"
        ↓
onChange event fires
        ↓
event.target.value = "Lam"
        ↓
setName("Lam")
        ↓
React re-renders
        ↓
<p>Hello, Lam!</p> updated on screen
```

Two-way Binding

- Typing → onChange calls setName()→ updates the state.
- React re-render → <input value={name}> displays the new value.

useEffect – Handling Side Effects

- Sometimes you need to interact with the outside world (fetch data, manipulate the DOM, set timers, etc.).
- These actions are called side effects
- useEffect() lets you perform side effects after rendering.

Case	Example	Description
Data Fetching	fetch("https://api.example.com/data")	Load data from an external API when component mounts.
DOM Manipulation	document.title = "New Title";	Update document title or access DOM nodes manually.
Timers / Intervals	setTimeout() / setInterval()	Schedule tasks or delayed updates.

useEffect – Handling Side Effects

```
useEffect(() => {
  // Effect logic here (runs after render)
  return () => {
    // Cleanup logic (optional)
  };
}, [deps]);
```

- Parameter 1: function (() => { ... })
  - Callback function that React runs after the component has been rendered.
  - It's used for: fetching data from an API, setting up event listeners, updating the document, etc.
- Parameter 2: The Dependency Array [deps]
  - Controls when the effect runs.
  - React compares the values in [deps] between renders: If any value changes → effect runs again.

useEffect – Handling Side Effects

Dependency Array	When It Runs	Common Use
(none)	After every render	DOM tracking, debugging
[]	Only once on mount	Fetch data, setup timer
[count]	When count changes	Title update, re-fetch
[count, name]	When any changes	Multi-condition effect

```
useEffect(() => {
  fetch("https://api.example.com/users")
    .then(res => res.json())
    .then(data => setUsers(data));
}, []); // run once on mount
```

[] → Empty dependency array means the effect runs only once

useEffect – Handling Side Effects

- "Only re-run this effect when the value of count is different from before."

```jsx
function CounterExample() {
  const [count, setCount] = React.useState(0);
  const [name, setName] = React.useState("Lam");

  React.useEffect(() => {
    console.log("Effect triggered: count =", count);
    document.title = `Count: ${count}`;
  }, [count]); // 👈 Only depends on 'count'

  return (
    <div>
      <h3>useEffect([count])</h3>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase Count</button>
      <p>Name: {name}</p>
      <button onClick={() => setName(name + "!")}>Change Name</button>
    </div>
  );
}
```

Action	Effect Runs?
First render	✅ Yes
Click "Increase Count"	✅ Yes
Click "Change Name"	❌ No

useEffect – Cleanup Function

- Prevent **memory leaks** and ensure a clean state before re-running the effect or unmounting component.
- cleanup function runs:
  - **Before** the component is unmounted.
  - **Before** each re-run of the effect (if dependencies change).

```
useEffect(() => {
  // Perform side effect
  const timer = setInterval(() => {
    console.log("Running...");
  }, 1000);

  // Cleanup before component unmounts or before next effect runs
  return () => {
    clearInterval(timer);
    console.log("Cleaned up!");
  };
}, []);
```

useEffect

```jsx
function FetchUsers() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  useEffect(() => {
    console.log("Fetching users...");
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(response => response.json())
      .then(data => {
        setUsers(data);
        setLoading(false);
      })
      .catch(error => console.error("Error fetching data:", error));
  }, []);
  if (loading) return <p>Loading users...</p>;
  return (
    <div>
      <h3>User List (Fetched with useEffect)</h3>
      <ul>
        {users.map(user => (
          <li key={user.id}>
            <strong>{user.name}</strong> - {user.email}
          </li>
        ))}
      </ul>
    </div>
  );
}
```

User List (Fetched with useEffect)

- **Leanne Graham** — Sincere@april.biz
- **Ervin Howell** — Shanna@melissa.tv
- **Clementine Bauch** — Nathan@yesenia.net

HUST

hust.edu.vn fb.com/dhbkhn

4. React Router & Context API

4.1. React Router

4.2. Context API

React Router

Feature	Traditional Website (MPA)	SPA (with React Router)
Page loading	Full page reload	Only content updates
Navigation	Handled by server	Handled by React Router
URL behavior	Real physical URLs	Virtual URLs (in browser history)
User experience	Slower transitions	Smooth, instant transitions

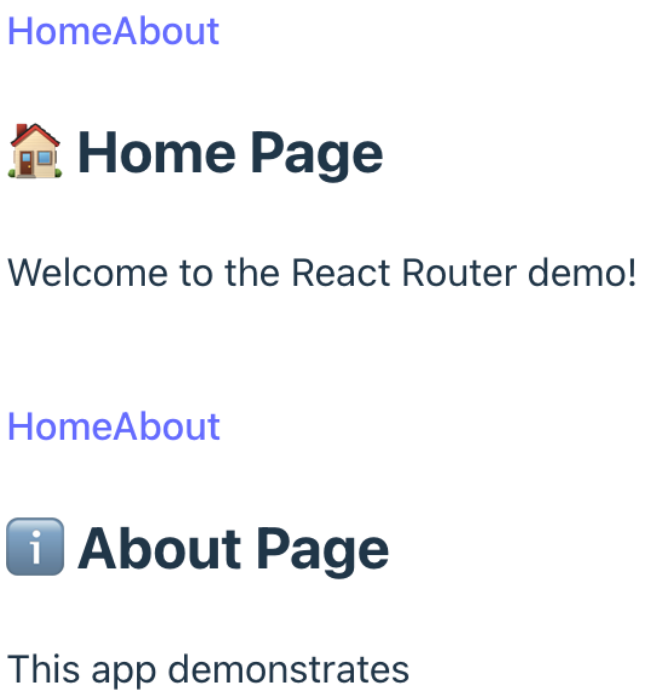
SPAs need a client-side routing system (like React Router) to manage different views or components based on the URL

React Router

- It maps each URL path (e.g., /home, /about) to a React Component (e.g., <Home />, <About />).
- No actual network request is sent — React just renders a different component.

```
return (
  <BrowserRouter>
    <nav style={{ marginBottom: "1rem" }}>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
    </nav>

    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
    </Routes>
  </BrowserRouter>
);
```



Installing React Router DOM

- react-router-dom: a React library that provides modules for routing in web applications.
- It contains many modules: BrowserRouter, Route, Link, etc.)
- Install: npm install react-router-dom
- Check: npm list react-router-dom

BrowserRouter & Routes

- BrowserRouter is a router component that keeps your UI in sync with the browser's address bar.
- BrowserRouter listens to URL changes → Routes display matching components

Function	Description
Root Wrapper	Encloses your entire app, enabling routing throughout it.
Manages History	Tracks navigation history (Back/Forward) like a real website.
Listens to URL Changes	Automatically updates components when the URL changes.
Prevents Full Reloads	Changes route client-side only, keeping app fast and seamless.

<Route> Component

- <Route> defines the mapping between a URL path and a React component.
- <Route path="/about" element={<About />} />

```
return (
  <BrowserRouter>
    <nav style={{ marginBottom: "1rem" }}>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
    </nav>

    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
    </Routes>
  </BrowserRouter>
);
```

Attribute	Description
path	Defines the URL pattern (e.g., /about, /users/:id).
element	Specifies which component should be rendered for that path.

URL Parameters & useParams

- URL parameters allow you to create dynamic routes that can capture variable parts of the URL.
- /users/:id

=> :id part is a parameter placeholder.

```
<Routes>
  <Route path="/users/:id" element={<UserProfile />} />
</Routes>
```

🧠 Meaning:

- When the URL is `/users/1` → show user #1
- When the URL is `/users/2` → show user #2

```
function UserProfile() {
  const { id } = useParams();
  return <h2>User ID: {id}</h2>;
}
```

Prop Drilling Problem

- Passing props through multiple layers that don't need them, just to reach the component that does.

```
App (has user data)
 └── Layout
      └── Content
           └── Sidebar
                └── UserProfile (uses user)
```

Issue	Description
Unnecessary coupling	Components become tightly connected.
Harder maintenance	Small changes require editing multiple files.
Poor scalability	Becomes messy as the app grows deeper.

Context API – Sharing Global Data

- Context API is a built-in React feature that allows you to share data globally across the component tree
- It provides a way to pass data through the component hierarchy directly — skipping intermediate components.

Context Type	Example
🎨 **Theme**	Light / Dark mode
🔐 **Auth**	Logged-in user, authentication state
🌐 **Language**	Multi-language settings (EN, VI, etc.)
⚙️ **Settings**	App configuration, preferences

Context API – Sharing Global Data

```
// 1️⃣ Create a Context
const ThemeContext = createContext("light");

// 2️⃣ Create a Provider
function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}

// 3️⃣ Consume the Context
function Toolbar() {
  return <Button />;
}

function Button() {
  const theme = useContext(ThemeContext);
  return <button>Current theme: {theme}</button>;
}
```

Step	Description
1	createContext() creates a global container for the shared data.
2	<ThemeContext.Provider> makes the data ("dark") available to all child components.
3	useContext(ThemeContext) allows any component to read the data directly

Key Concepts Recap

Concept	Definition	Key Idea / Example
Virtual DOM (VDOM)	A lightweight, in-memory representation of the real DOM.	React compares old and new VDOMs → updates only changed parts.
Component	Reusable building block of the UI (function or class).	<Button />, <Header /> — combines structure, logic, and style.
State	Internal, mutable data of a component.	Managed with useState() → triggers re-render on update.
Props	Read-only data passed from parent to child component.	<User name="Alice" /> → child uses props.name.
Hooks	Special functions that let functional components use state and lifecycle features.	useState, useEffect, useContext, etc.
Router	Manages navigation and URL paths without page reload.	react-router-dom: <BrowserRouter>, <Route>, <Link>.
Context	Global data sharing system that avoids prop drilling.	Theme, Auth, or Language shared via <Context.Provider>.