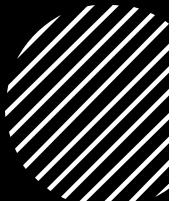




Gautam Singh
React JS



var



Function scoped



Can be re-declared & re-assigned



Hoisted (initialized as undefined)



```
var x = 10;  
var x = 20; // allowed  
x = 30; // allowed  
console.log(x); // 30
```

let

- Block scoped
- Can be re-assigned
- Cannot be re-declared in same scope
- `let y = 10;`
- `y = 20; // allowed`
- `// let y = 30; ✗ error`
- `console.log(y); // 20`

const

- Block scoped
- Cannot be re-assigned or re-declared
- Must be initialized
- `const z = 100;`
- `// z = 200; ✗ error`
-
- `const obj = { name: "John" };`
- `obj.name = "Doe"; // ✓ allowed (object mutation)`



Normal Function

- `function add(a, b) {`
 - `return a + b;`
 - }
-

Arrow Function

- `const add = (a, b) => a + b;`

With Single Parameter

- `const square = x => x * x;`

Arrow Function & this

- const user = {
- name: "Gautam",
- greet: () => {
- console.log(this.name); // ✗ undefined
- }
- };

Destructuring ({} , [])

- Object Destructuring

- const user = {
 - name: "Gautam",
 - age: 25,
 - city: "Delhi"
- };
-
- const { name, age } = user;
- console.log(name, age);



Rename Variables

- `const { name: username } = user;`
 - `console.log(username);`
-



Array Destructuring

- const numbers = [10, 20, 30];
 -
 - const [a, b] = numbers;
 - console.log(a, b); // 10 20
-

- Spread Operator
- Used to **copy or merge**

- `const arr1 = [1, 2];`
- `const arr2 = [...arr1, 3, 4];`
- `console.log(arr2);`
-
- `const obj1 = { a: 1 };`
- `const obj2 = { ...obj1, b: 2 };`

Rest Operator

- Used in **functions**
- ```
function sum(...numbers) {
```
- `return numbers.reduce((a, b) => a + b);`
- }
- 
- `console.log(sum(1, 2, 3)); // 6`

- Array Methods: map, filter, reduce
- map() – Transform Array

```
• const nums = [1, 2, 3];
• const squared = nums.map(n => n * n);
• console.log(squared); // [1,4,9]
```



# filter() - Condition Based

- `const nums = [1, 2, 3, 4];`
  - `const even = nums.filter(n => n % 2 === 0);`
  - `console.log(even); // [2,4]`
-

# reduce() – Single Value

- const nums = [1, 2, 3, 4];
- 
- const sum = nums.reduce((total, num) => total + num, 0);
- console.log(sum); // 10

# Callback Functions

- A function passed as an argument.
- ```
function greet(name, callback) {
```
- ```
 console.log("Hello " + name);
```
- ```
    callback();
```
- ```
}
```
- 
- ```
function sayBye() {
```
- ```
 console.log("Bye!");
```
- ```
}
```
-
- ```
greet("Gautam", sayBye);
```

- Promises & `async / await`
- Promise Example

- `const promise = new Promise((resolve, reject) => {  
 let success = true;  
 if (success) {  
 resolve("Data fetched");  
 } else {  
 reject("Error occurred");  
 }  
});  
  
promise  
.then(data => console.log(data))  
.catch(err => console.log(err));`

# Async / Await (Cleaner)

```
• async function fetchData() {
• try {
• const response = await promise;
• console.log(response);
• } catch (error) {
• console.log(error);
• }
• }
•
• fetchData();
```

- +
- 

- Closure
- Function remembers its outer scope.

- function outer() {
- let count = 0;
- function inner() {
- count++;
- console.log(count);
- }
- return inner;
- }
- const counter = outer();
- counter(); // 1
- counter(); // 2

# Hoisting

## Variable/function moved to top of scope.

```
console.log(a); // undefined
var a = 10;
```

```
hello(); // works
function hello() {
 console.log("Hello");
}
```

**let & const** are hoisted but in **temporal dead zone**

# Export/Import

- // math.js

```
export const add = (a, b) => a + b;
export const sub = (a, b) => a - b;
```

```
import { add, sub } from "./math.js";
console.log(add(2, 3));
```

# Default Export



- // user.js
- export default function user()  
{
- console.log("User");
- }
- import user from "./user.js";

# Spread Operator

( . . . )



Expands elements



```
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5];

console.log(arr2); // [1,2,3,4,5]
```



```
const user = { name: "Gautam" };
const userDetails = { ...user, age: 25 };

console.log(userDetails);
```

# Rest Operator (...)

- → **Collects elements into an array**
- ```
function sum(...numbers) {
```
- `return numbers.reduce((a, b) => a + b, 0);`
- }
-
- `console.log(sum(1, 2, 3, 4)); // 10`

Difference between map() and forEach()

Feature	map	forEach
Returns new array	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Used for	Transformation	Side effects
Chainable	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Mutates original	<input type="checkbox"/> No	<input type="checkbox"/> No

map()

```
const nums = [1, 2, 3];  
const doubled = nums.map(n => n * 2);
```

```
console.log(doubled); // [2,4,6]
```

forEach()

```
const nums = [1, 2, 3];  
  
nums.forEach(n => {  
  console.log(n * 2);  
});
```

- Callback Hell
- Problem :-
- Hard to read
- Hard to maintain
 - Error handling is difficult

- setTimeout(() => {
- console.log("Step 1");
- setTimeout(() => {
- console.log("Step 2");
- setTimeout(() => {
- console.log("Step 3");
- }, 1000);
- }, 1000);
- }, 1000);

Solution Using Promises

```
function step(msg, time) {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      console.log(msg);  
      resolve();  
    }, time);  
  });  
}  
  
step("Step 1", 1000)  
.then(() => step("Step 2", 1000))  
.then(() => step("Step 3", 1000));
```

Why async/await is better than Promises?

- fetchData()
- .then(data => {
- return processData(data);
- })
- .then(result => {
- console.log(result);
- })
- .catch(error => console.log(error));

Async/Await Code

- async function getResult() {
- try {
- const data = await fetchData();
- const result = await processData(data);
- console.log(result);
- } catch (error) {
- console.log(error);
- }
- }
- getResult();

What is happening? / Promise

fetchData()
returns a **Promise**

.then() waits for
the promise to
resolve

Result is passed to
next .then()

.catch() handles
errors

✖ Problems

Nested .then()
becomes hard to
read

Error handling is
less intuitive

Looks
asynchronous and
complex



Interview
Line:

Promise chaining
works but reduces
readability as
code grows.

What is happening? `async/await`

async function
always returns a
Promise

await pauses
execution until
Promise resolves

Code executes **top to bottom**

try/catch handles
errors clearly

✓ Benefits

Cleaner and readable

Looks synchronous

Easy debugging

Better error handling

➔ Interview Line:

Async/await
simplifies promise-based code and improves readability.

SIDE-BY-SIDE COMPARISON (IMPORTANT)

Feature	Promise .then ()	Async/Await
Readability	✗ Average	✓ Excellent
Error handling	.catch ()	try/catch
Debugging	✗ Hard	✓ Easy
Nesting	✗ Complex	✓ Flat
Modern usage	✗ Less	✓ Preferred

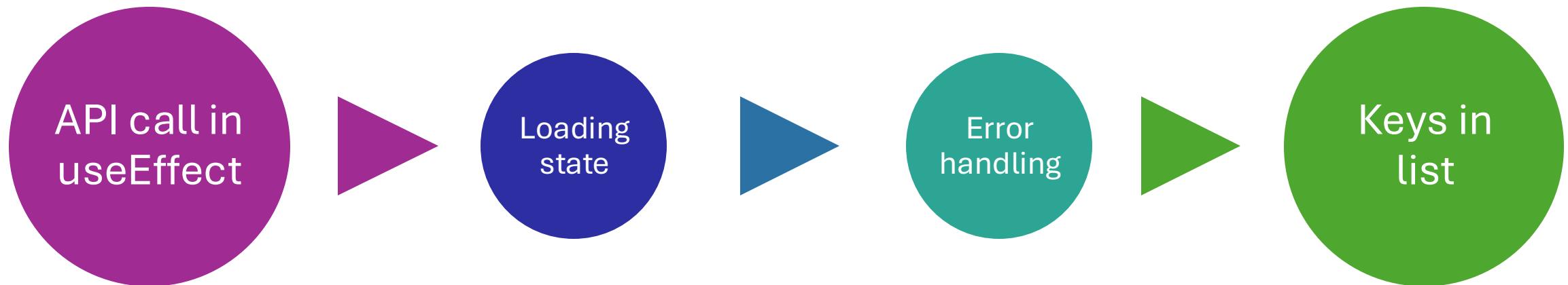
Fake API in React Component

- import { useEffect, useState } from "react";
- function Users() {
- const [users, setUsers] = useState([]);
- const [loading, setLoading] = useState(true);
- useEffect(() => {
- fetch("https://jsonplaceholder.typicode.com/users")
- .then(res => res.json())
- .then(data => {
- setUsers(data);
- setLoading(false);
- })
- .catch(err => {
- console.log(err);
- setLoading(false);
- });
- }, []);

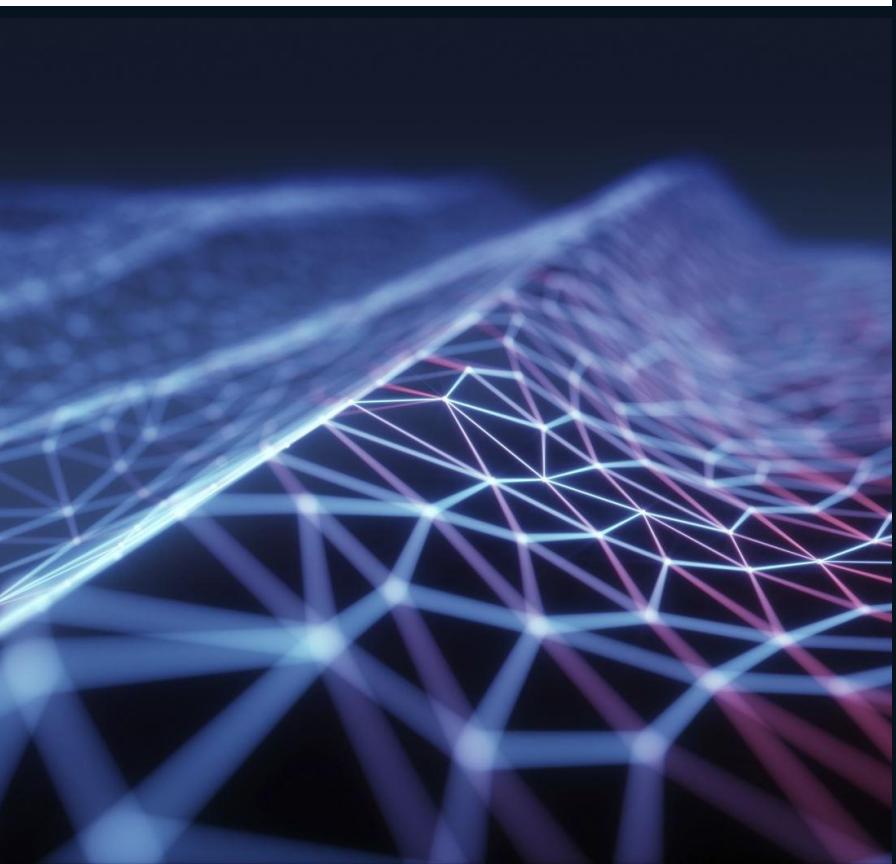
Previous Page Code

- if (loading) return <h2>Loading...</h2>;
- return (
-
- {users.map(user => (
- <li key={user.id}>{user.name}
-))}
-
-);
- }
- export default Users;

Interview Questions Covered



Fake API using Axios



```
import axios from "axios";  
  
async function getData() {  
  
  try {  
  
    const response = await axios.get(  
      "https://jsonplaceholder.typicode.com/posts"  
    );  
  
    console.log(response.data);  
  
  } catch (error) {  
  
    console.log(error);  
  
  }  
  
}  
  
getData();
```

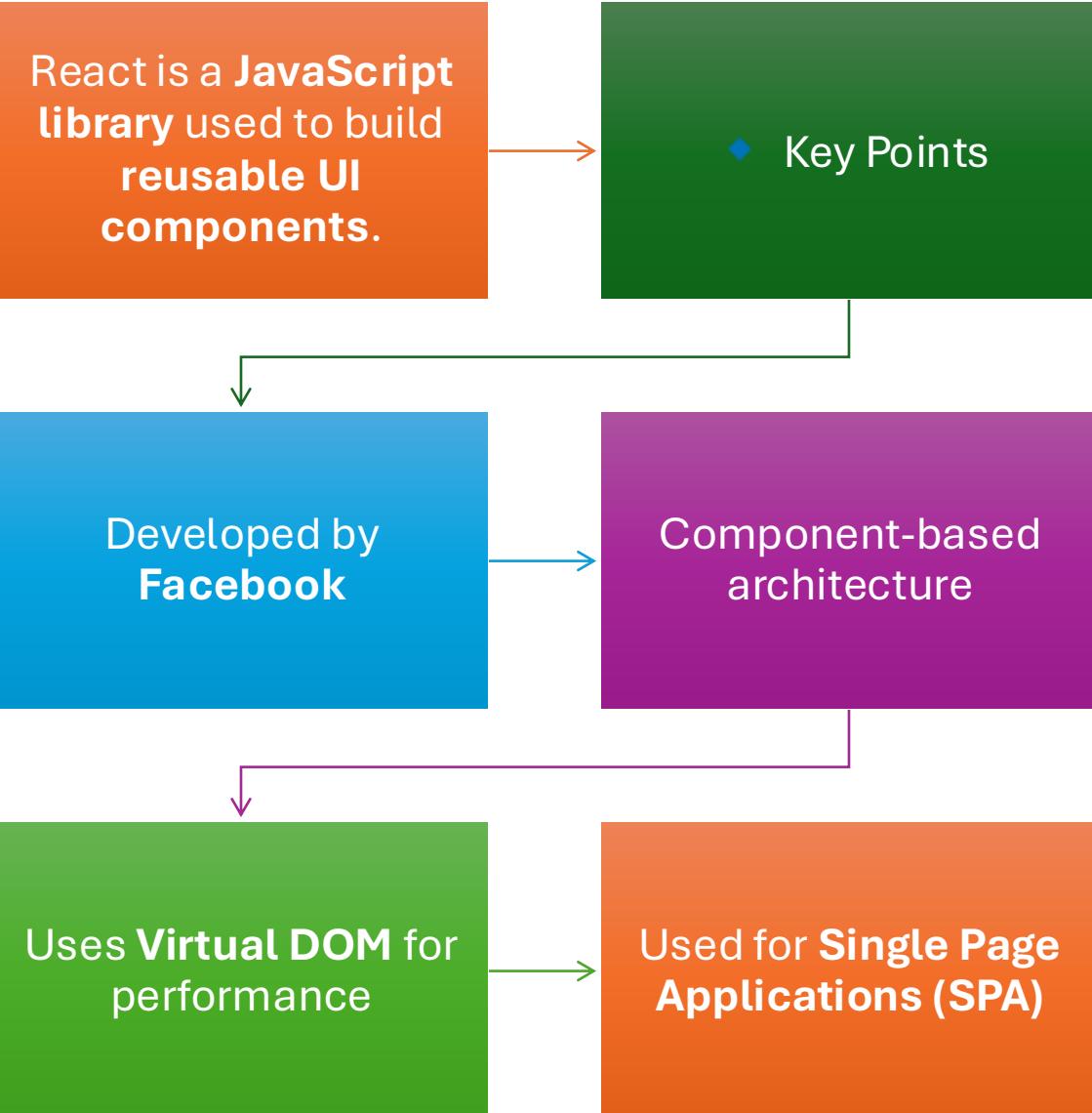
Why Axios preferred?

Auto JSON
parsing

Interceptors

Better error
handling

What is React?



Why React is fast? (VERY IMPORTANT)

React is fast because:

Uses **Virtual DOM** instead of real DOM

Diffing algorithm updates only changed parts

Uses **batch updates**

Reusable components reduce re-render cost

📌 **One-line Interview Answer:**

React is fast because it minimizes direct DOM manipulation using Virtual DOM.

JSX (JavaScript XML)

What is JSX?

JSX allows writing **HTML-like code inside JavaScript**.

```
const element = <h1>Hello  
World</h1>;
```

Behind the Scenes

```
React.createElement("h1", null, "Hello  
World");
```

Is JSX mandatory?

No — React can work without JSX.



Interview Answer:

JSX is optional but improves readability and developer experience.

What is a Component?

A component is a **reusable piece of UI**.

Functional Component (Preferred)

```
function Welcome() {
```

```
    return <h1>Hello</h1>;
```

```
}
```

```
export default Welcome;
```

Class Component (Older)

```
class Welcome extends React.Component {
```

```
    render() {
```

```
        return <h1>Hello</h1>;
```

```
    }
```

```
}
```

Why functional components are preferred?

Reasons

Less code

Easier to read

Hooks available

Better performance

No this keyword confusion



One-line Interview Answer:

Functional components are preferred because they are simpler, cleaner, and support hooks.

What are Props?

- Props are **inputs passed from parent to child**.

- Read-only
- Cannot be modified by child

- function Parent() {
 • return <Child name="Gautam" />;
• }

- function Child({ name }) {
 • return <h1>Hello {name}</h1>;
• }

Difference between Props and State?

Props	State
Passed from parent	Managed inside component
Read-only	Mutable
Cannot change	Can change
Used for configuration	Used for data management

What is State?

- import { useState } from "react";
- function Counter() {
- const [count, setCount] = useState(0);
- return (
 - <>
 - <h2>Count: {count}</h2>
 - <button onClick={() => setCount(count + 1)}>
 - Increment
 - </button>
 - </>
-);
- }
- export default Counter;

Why state updates are async?

Improves performance

Allows batching of updates

Prevents unnecessary re-renders



One-line Interview Answer:

State updates are async to optimize performance through batching.

What are Hooks?

Hooks allow **functional components** to use **state** and **lifecycle features**.

Common Hooks with COMPLETE Code

1. useState – Manage State

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

2. useEffect – Side Effects

Mounting Example

```
import { useEffect } from "react";
```

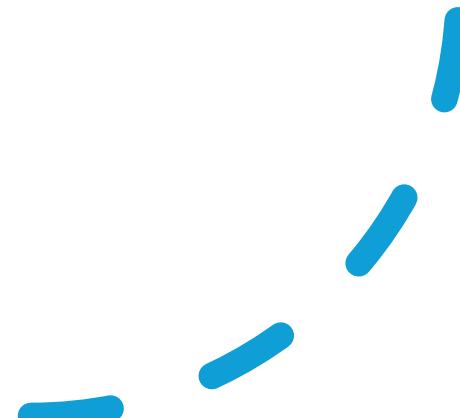
```
useEffect(() => {
```

```
    console.log("Component mounted");
```

```
}, []);
```

Common Hooks with COMPLETE Code

- `useEffect(() => {`
- `return () => {`
- `console.log("Component unmounted");`
- `};`
- `}, []);`
- **Used for:**
- API calls
- Event listeners
- Timers



+

•

○

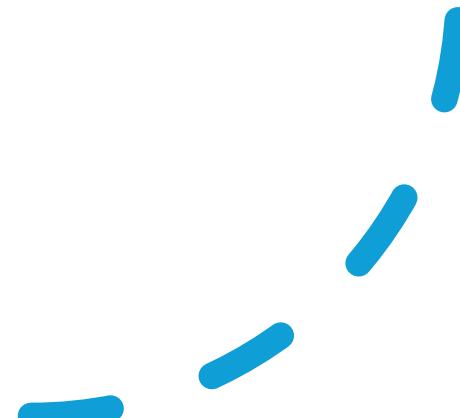
Common Hooks with `useContext` - Global State

- `const ThemeContext = createContext();`
- `function App() {`
- `return (`
- `<ThemeContext.Provider value="dark">`
- `<Child />`
- `</ThemeContext.Provider>`
- `);`
- `}`

- `function Child() {`
- `const theme = useContext(ThemeContext);`
- `return <h1>{theme}</h1>;`
- `}`

Common Hooks with useRef – DOM Access

- import { useRef } from "react";
- function InputFocus() {
- const inputRef = useRef(null);
-
- function focusInput() {
- inputRef.current.focus();
- }
-
- return (
- <>
- <input ref={inputRef} />
- <button onClick={focusInput}>Focus</button>
- </>
-);
- }



Common Hooks with useMemo – Optimization

- import { useMemo } from "react";
- const expensiveCalculation = num => {
 - console.log("Calculating...");
 - return num * 2;
 - };
- const result = useMemo(() => expensiveCalculation(count), [count]);

Common Hooks with useCallback – Performance



```
import { useCallback } from "react";
```



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

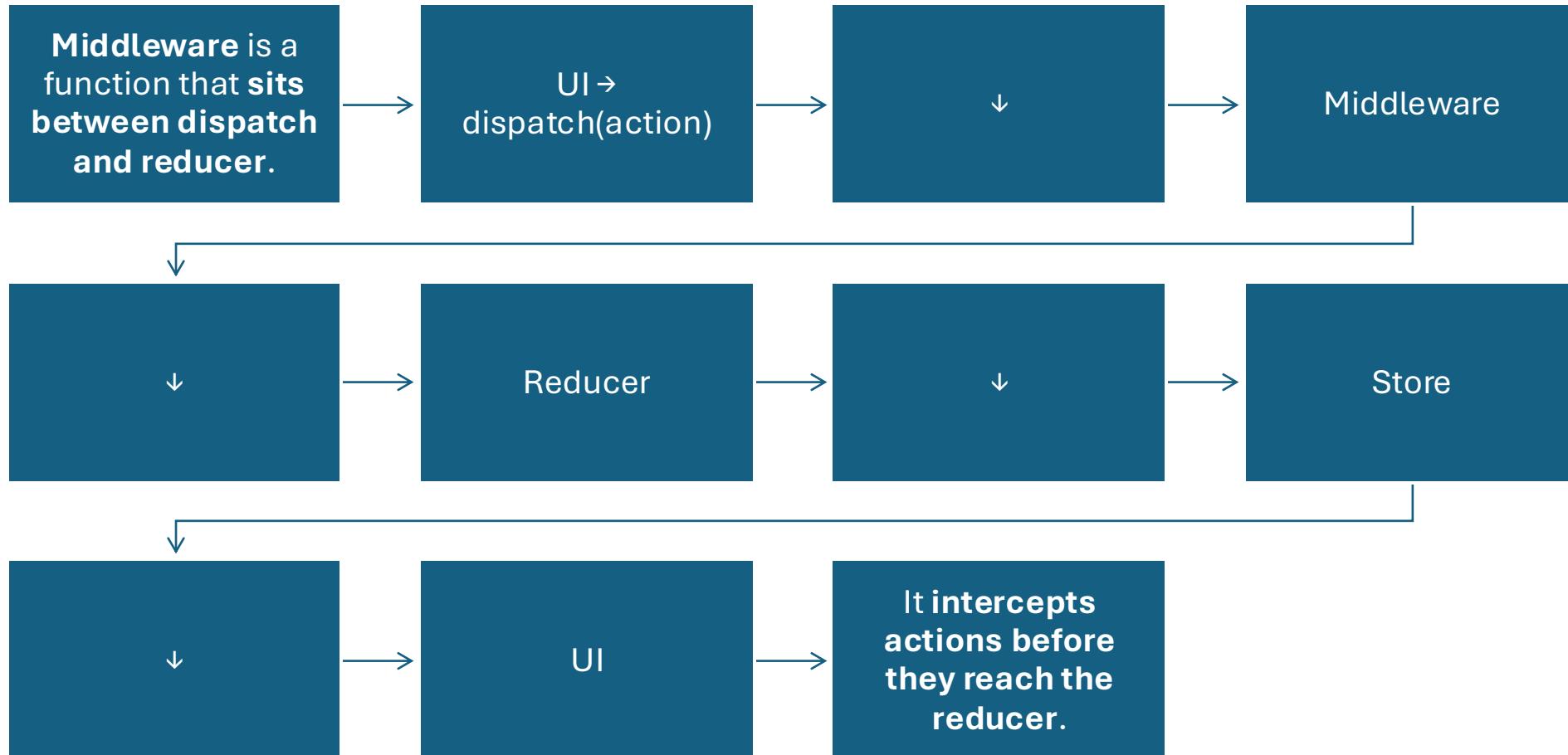


```
    console.log("Clicked");
```



```
}, []);
```

What is Middleware?



What is Redux?

Redux is a **state management library** used to manage **global application state** in a **predictable way**.

- ◆ Why Redux?

- Avoids **prop drilling**

- Centralized state

- Easy debugging

- Predictable state updates

📌 **Interview One-liner:**

Redux manages global state in a predictable, centralized store.

When Should You Use Redux?

Large applications

- ✓ Shared state across many components
- ✓ Complex state logic

✗ Small apps

✗ Single component state



Interview Trick Question:



Redux is not mandatory in React.

Core Redux Concepts (MOST IMPORTANT)

1. Store

Single source of truth

Holds application state

```
const store =  
  createStore(reducer);
```

2. Action

Plain JavaScript object

Must have type

{ type: "INCREMENT" }

3. Reducer

```
function counterReducer(state = { count: 0 }, action) {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    default:
      return state;
  }
}
```

4. Dispatch

Sends action to reducer

```
store.dispatch({  
  type:  
  "INCREMENT"  
});
```

What is React Lifecycle?

The **React lifecycle** is the series of stages a component goes through from:

👉 creation → update → removal

THREE MAIN PHASES

- **1 Mounting** – Component is created and inserted into DOM
- **2 Updating** – Component re-renders due to changes
- **3 Unmounting** – Component is removed from DOM

1 MOUNTING PHASE

What
Happens?

Component
is created

JSX rendered

DOM updated

Side effects
run (API calls,
events)



2

UPDATING PHASE

- When does update happen?
 - State change (`setState`, `useState`)
 - Props change
-

3

UNMOUNTING PHASE

What Happens?

Component
removed from
DOM

Cleanup tasks
run

1

Mounting Phase

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

```
function LifecycleDemo() {
```

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

```
// 1 Mounting Phase
```

```
  useEffect(() => {
```

```
    console.log(" Component Mounted");
```

```
    return () => {
```

```
      console.log(" Component Unmounted");
```

```
    };
```

```
  }, []);
```

2 Updating Phase

```
// [2] Updating Phase
useEffect(() => {
  console.log("🟡 Component Updated - Count:", count);
}, [count]);
return (
<div style={{ padding: "20px", border: "2px solid black" }}>
<h2>React Lifecycle Demo</h2>
<h3>Count: {count}</h3>
<button onClick={() => setCount(count + 1)}>
  Update State
</button>
</div>
);
}
export default LifecycleDemo;
```

Parent Component (To Show Unmounting)

```
• import { useState } from "react";
• import LifecycleDemo from "./LifecycleDemo";
• function App(){
•   const [show, setShow] = useState(true);
•   return (
•     <div style={{ padding: "20px" }}>
•       <button onClick={() => setShow(!show)}>
•         {show ? "Unmount Component" : "Mount Component"}
•       </button>
•       <br /><br />
•       {show && <LifecycleDemo />}
•     </div>
•   );
• }
• export default App;
```