ReactJS

Why React Was Created

React was developed by Facebook (now Meta) and released as an open-source project in 2013. It was created to solve specific problems:

1. **Performance issues with complex UIs**: Facebook was struggling with their notification system as they had a lot of dynamic UI updates.
2. **Code maintainability challenges**: As web applications grew more complex, keeping the UI in sync with the data became increasingly difficult.
3. **The need for component reusability**: They wanted a way to build encapsulated components that manage their own state.

Core Benefits of React

1. **Declarative approach**: You describe what your UI should look like for different states, and React efficiently updates and renders the right components when data changes.
2. **Component-based architecture**: Everything in React is a component, which makes code more reusable, maintainable, and easier to test.
3. **Virtual DOM**: React creates a lightweight representation of the real DOM in memory, which improves performance by minimising direct DOM manipulation.
4. **Unidirectional data flow**: Data flows down from parent to child components, making the application more predictable and easier to debug.
5. **Rich ecosystem**: React has a large community and ecosystem of libraries, tools, and extensions.

▼ Fundamentals

- **Components**: Reusable, self-contained pieces of code that return React elements

```
// Functional component
function Welcome() {
  return <h1>Hello, React!</h1>;
}
```

- **Props**: Read-only data passed from parent to child components

```
// Parent component
function App() {
  return <Welcome name="Sara" />;
}

// Child component
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

- **State**: Mutable data managed within a component

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

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

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
```

```
  );
}
```

▼ JSX Syntax

```
// JSX expressions
const name = 'Josh';
const element = <h1>Hello, {name}</h1>;

// Multiple attributes
const element = (
  <img
    src={user.avatarUrl}
    className="profile-pic"
    alt="User profile"
  />
);

// Must have one parent element or fragment
return (
  <>
    <h1>Title</h1>
    <p>Content</p>
  </>
);
```

▼ Creating Components

```
// Creating a functional component
import React from 'react';

const Profile = ({ name, bio }) => {
  return (
```

```
    <div className="profile">
      <h2>{name}</h2>
      <p>{bio}</p>
    </div>
  );
};

// Rendering the component
import Profile from './Profile';

function App() {
  return <Profile name="John" bio="React Developer" />;
}
```

~~Component Lifecycle Methods~~

~~This section primarily applies to class components~~

Handling Events

```
function ActionButton() {
  const handleClick = (text) => {
    alert(text);
  };

  return (
    <button onClick={() => handleClick('Button clicked!')}>
      Click Me
    </button>
  );
}
```

Conditional Rendering

```
function Greeting({ isLoggedIn }) {
// Using && operator
  return (
    <div>
      {isLoggedIn && <h1>Welcome back!</h1>}
      {!isLoggedIn && <h1>Please sign in</h1>}

      {/* Alternative: using ternary operator */}
      {isLoggedIn ? (
        <button>Logout</button>
      ) : (
        <button>Login</button>
      )}
    </div>
  );
}
```

▼ Lists and Keys

```
function TodoList({ todos }) {
  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          {todo.text}
        </li>
      ))}
    </ul>
  );
}

// Usage
const todos = [
```

```
  { id: 1, text: 'Learn React' },
  { id: 2, text: 'Build an app' }
];
<TodoList todos={todos} />
```

▼ Forms and Controlled Components

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

function Form() {
  const [name, setName] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Form submitted with name:', name);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}
```

▼ React Developer Tools

- Browser extension to inspect components

- Example usage: Right-click on element → Inspect → React tab
- View component hierarchy, props, state, and hook values
- Performance profiler for measuring render times and identifying bottlenecks

Core React Concepts

▼ Class vs. Function components

Feature	Functional Components	Class Components
State Management	It can use Hooks like useState, useReducer	It uses this.state and this.setState()
Lifecycle Methods	It uses **useEffect** Hook for lifecycle methods	It uses traditional lifecycle methods like componentDidMount, componentWillUnmount
Rendering	Returns JSX directly inside the function	Uses a render() method to return **JSX**
Performance	Faster and more lightweight due to simpler structure	Slightly heavier due to the overhead of class instances
Hooks	Can use React hooks (**useState**, useEffect, etc.)	Cannot use hooks; relies on lifecycle methods and state
This Keyword	Does not use this keyword	Uses this to access props and state
Code Complexity	Less boilerplate code, easier to write and understand	More boilerplate code, especially for state and methods
Event Handling	Simple and direct event handling	Requires method binding for event handling

▼ React Hooks

Core Hooks

1. **useState** - Manages state in functional components, returns a stateful value and a function to update it.

2. **useEffect** - Handles side effects in components, such as data fetching, subscriptions, or DOM manipulation.
3. **useContext** - Accepts a context object and returns the current context value from the nearest provider.
4. **useReducer** - Alternative to useState for complex state logic, uses a reducer function similar to Redux.
5. **useCallback** - Returns a memoized callback function that only changes if dependencies change, preventing unnecessary re-renders.
6. **useMemo** - Returns a memoized value that only recalculates when dependencies change, optimizing expensive calculations.
7. **useRef** - Returns a mutable ref object that persists for the lifetime of the component, useful for accessing DOM elements directly.
8. **useImperativeHandle** - Customizes the instance value exposed when using forwardRef, useful for parent components controlling child component methods.
9. **useLayoutEffect** - Similar to useEffect but fires synchronously after DOM mutations and before browser paint, useful for DOM measurements.
10. **useDebugValue** - Used for displaying a label for custom hooks in React DevTools for debugging.

Additional Hooks in React 18+

1. **useTransition** - Marks state updates as non-urgent transitions, allowing urgent updates to interrupt them.
2. **useDeferredValue** - Defers updating a non-critical part of the UI, similar to debouncing.
3. **useId** - Generates unique IDs for accessibility attributes, useful for server rendering.
4. **useSyncExternalStore** - Subscribes to external stores safely, ensuring consistency during concurrent rendering.

5. **useInsertionEffect** - Similar to useLayoutEffect but fires before any DOM mutations, designed for CSS-in-JS libraries.

React Router Hooks

1. **useParams** - Returns an object of dynamic parameters from the current URL.
2. **useNavigate** - Returns a navigation function for programmatic navigation.
3. **useLocation** - Returns the current location object with information about the URL.
4. **useRoutes** - Used to define routes programmatically.
5. **useSearchParams** - Used to read and modify the query string in the URL.

React Query Hooks

1. **useQuery** - Fetches, caches, and manages server state with automatic refetching.
2. **useMutation** - Handles create/update/delete operations and server state mutations.
3. **useQueryClient** - Provides access to the QueryClient to interact with the cache.

React Redux Hooks

- **useSelector** - Extracts data from the Redux store state
- **useDispatch** - Returns the Redux store's dispatch method for dispatching actions
- **useStore** - Gives direct access to the Redux store

React Form Hooks (from React Hook Form)

- **useForm** - The main hook for form state management and validation
- **useController** - For creating controlled inputs

- **useFormContext** - Accesses form context when working with nested components
- **useWatch** - Watches specific form values

React Testing Library Hooks

- **renderHook** - Helper for testing custom hooks

▼ Context API

What is Context API?

The Context API is a feature in React that provides a way to share data between components without having to explicitly pass props through every level of the component tree. It's essentially a global state management system built into React.

Why We Use Context API

1. **Avoid Prop Drilling**: Eliminates the need to pass props through intermediate components that don't need the data but only serve as a means to pass it down.
2. **Global State Management**: Provides a way to manage global state such as user authentication, themes, language preferences, etc.
3. **Component Reusability**: Makes components more reusable as they can connect directly to the context they need.
4. **Simplified Data Flow**: Creates a more direct data flow for certain types of application data.

How to Use Context API

Step 1: Create a Context

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

```
// Create a context with optional default value
const ThemeContext = createContext('light');
```

Step 2: Create a Provider Component

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

function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {
    setTheme(theme === 'light' ? 'dark' : 'light');
  };

  // Value can be any data type - objects are common
  const value = {
    theme,
    toggleTheme
  };

  return (
    <ThemeContext.Provider value={value}>
      {children}
    </ThemeContext.Provider>
  );
}
```

Step 3: Wrap Your Application

```
function App() {
  return (
    <ThemeProvider>
      <MainContent />
    </ThemeProvider>
```

```
  );
}
```

Step 4: Consume the Context

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

function ThemedButton() {
  // Access context value with useContext
  const { theme, toggleTheme } = useContext(ThemeContext);

  return (
    <button
      onClick={toggleTheme}
      style={{
        background: theme === 'light' ? '#fff' : '#333',
        color: theme === 'light' ? '#333' : '#fff'
      }}
    >
      Toggle Theme
    </button>
  );
}
```

Advantages of Context API

1. **Built-in Solution**: No need for external libraries for simple global state management.
2. **Performance Optimization**: Context can be split into multiple smaller contexts to prevent unnecessary re-renders.
3. **Simplicity**: Easier to set up and understand compared to more complex state management libraries.
4. **Integration with Hooks**: Works seamlessly with other React hooks.

5. **Reduced Component Complexity**: Components don't need to manage passing props they don't use.
6. **Flexible Updates**: Context can be updated from any component that has access to the update function.
7. **TypeScript Support**: Well-supported in TypeScript for type safety.

Limitations to Be Aware Of

1. Not ideal for high-frequency updates (though this has improved in recent React versions)
2. Can make component reuse more difficult if they're tightly coupled to specific contexts
3. Might lead to performance issues if context is too large or changes too frequently
4. Doesn't provide built-in middleware or devtools like dedicated state management libraries

Context API is perfect for global themes, user data, localization, and other application-wide settings, but for complex state management with many interactions, you might still consider libraries like Redux or Zustand.

▼ Refs and the DOM

React Refs

Purpose: Direct access to DOM elements or React components

Key Points:

- Use when you need to access DOM elements directly
- Avoid overusing refs (prefer declarative approach when possible)
- Common use cases: focus management, animations, third-party DOM libraries

Creating Refs:

```
// Using useRef hook (functional components)
const inputRef = useRef(null);

// Accessing the element
inputRef.current.focus();
```

Example - Focus an Input:

```
function AutoFocusInput() {
  const inputRef = useRef(null);

  useEffect(() => {
    // Focus the input when component mounts
    inputRef.current.focus();
  }, []);

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

React and the DOM

Virtual DOM: React's in-memory representation of the real DOM

Key Concepts:

- React handles DOM updates efficiently through reconciliation

  ▼ what is Reconciliation?

  Reconciliation is one of React's core concepts - it's the algorithm React uses to update the DOM efficiently when your application's state changes.

How Reconciliation Works

1. **Virtual DOM Comparison**

- When state or props change, React creates a new virtual DOM tree
- It then compares this new tree with the previous virtual DOM tree
- React identifies exactly what has changed between the trees

2. **Diffing Algorithm**
   - React uses a "diffing" algorithm to efficiently determine the minimal changes needed
   - It works on key assumptions that optimize performance:
     - Different component types produce different tree structures
     - Elements with stable keys maintain their identity across renders
3. **Batch Updates**
   - React batches all the necessary DOM updates together
   - This minimizes browser reflows and repaints

Key Reconciliation Rules

Element Type Comparison: If element types are different, React rebuilds the entire subtree

```
// Before
<div>
  <Counter />
</div>

// After (Counter replaced by Timer - entire subtree rebuilds)
<div>
  <Timer />
</div>
```

Keys for List Items: Keys help React identify which items have changed, been added, or removed

```
// Good: Each item has a stable, unique key
function TodoList({ todos }) {
  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>{todo.text}</li>
      ))}
    </ul>
  );
}
```

Performance Implications

- Without keys in lists, React might re-render the entire list when one item changes
- Component type changes trigger full subtree rebuilds (costly for deep components)
- React only updates the parts of the DOM that actually changed, not the entire page

Example of Reconciliation Process

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

  return (
    <div>
      <h1>Counter</h1>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</butto
n>
```

```
    </div>
  );
}
```

When the button is clicked:

1. State changes from `count: 0` to `count: 1`
2. React creates a new virtual DOM tree
3. React compares the trees and identifies only the text content in the `<p>` element has changed
4. React updates just that text node in the actual DOM, not the entire component

This selective updating is what makes React efficient for complex UIs with frequent changes.

- Direct DOM manipulation should be avoided when possible
- React updates the real DOM only where needed

Example - Measuring Element Height:

```
function MeasureExample() {
  const [height, setHeight] = useState(0);
  const divRef = useRef(null);

  useEffect(() => {
    if (divRef.current) {
      setHeight(divRef.current.clientHeight);
    }
  }, []);

  return (
    <>
      <div ref={divRef}>Measure this content</div>
      <p>The above div is {height}px tall</p>
    </>
```

```
  );
}
```

Forwarding Refs:

```
// For passing refs through components
const FancyButton = forwardRef((props, ref) => (
  <button ref={ref} className="fancy-btn">
    {props.children}
  </button>
));

// Usage
function Parent() {
  const buttonRef = useRef(null);

  const handleClick = () => {
    buttonRef.current.focus();
  };

  return (
    <>
      <FancyButton ref={buttonRef}>Click me!</FancyButton>
      <button onClick={handleClick}>Focus the fancy button</button>
    </>
  );
}
```

▼ **Interview questions:**

- Explain when to use refs vs state
- Discuss potential issues with direct DOM manipulation
- Demonstrate understanding of the virtual DOM concept
- Show awareness of useRef hook (for functional components)

▼ When to Use Refs vs State

Use State when:

- The data affects rendering and UI
- Information needs to trigger re-renders when changed
- Values need to be reactive across component tree
- Example: Form input values that display in UI

```
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button
>
    </div>
  );
}
```

Use Refs when:

- You need DOM access without triggering re-renders
- Storing values that persist between renders but don't affect UI
- Accessing/controlling DOM elements directly
- Example: Focus management, measurements, animations

```
function FocusInput() {
  const inputRef = useRef(null);

  return (
    <div>
      <input ref={inputRef} />
      <button onClick={() => inputRef.current.focus()}>Focus Input</
```

```
button>
    </div>
  );
}
```

▼ Potential Issues with Direct DOM Manipulation

1. **Breaking React's Unidirectional Data Flow**
    - React expects UI updates through state/props
    - Direct DOM changes can cause sync issues between React's virtual DOM and actual DOM
2. **Reduced Code Maintainability**
    - Makes debugging harder
    - Creates dependencies between DOM structure and JavaScript logic
3. **Component Reusability Problems**
    - Components with direct DOM manipulation are less portable
    - Often tied to specific DOM structures
4. **Performance Concerns**
    - Bypasses React's optimized rendering system
    - May cause unnecessary re-renders or flickering
5. **Testing Difficulties**
    - Components with direct DOM manipulation are harder to test
    - May require DOM simulation in test environment

▼ Understanding the Virtual DOM

- **What it is:** In-memory representation of the real DOM
- **How it works:**

1. React builds and maintains a lightweight copy of the DOM
2. When state changes, React creates a new virtual DOM tree
3. React compares previous and new virtual DOM (diffing)
4. Only necessary changes are applied to the real DOM (reconciliation)

- **Benefits:**
  - Batches DOM updates for better performance
  - Minimizes direct manipulation of the slow real DOM
  - Allows for cross-platform rendering (web, mobile, etc.)

▼ useRef Hook in Functional Components

- **Purpose:** Creates mutable object with .current property persisting across renders
- **Key Characteristics:**
  - Changes to ref.current don't trigger re-renders
  - Value persists for component's lifetime
  - Can store any value, not just DOM references

```
function Stopwatch() {
  const [time, setTime] = useState(0);
  // Store intervalId without causing re-renders
  const intervalRef = useRef(null);

  const start = () => {
    intervalRef.current = setInterval(() => {
      setTime(t => t + 1);
    }, 1000);
  };
```

```
  const stop = () => {
    clearInterval(intervalRef.current);
  };

  return (
    <div>
      <p>Time: {time}s</p>
      <button onClick={start}>Start</button>
      <button onClick={stop}>Stop</button>
    </div>
  );
}
```

- **Common useRef Applications:**
  - DOM element access
  - Previous value storage
  - Instance variables in functional components
  - Managing timers and intervals

- Error boundaries (Class component)

▼ Higher-Order Components (HOCs)

HOCs are functions that take a component and return a new enhanced component with additional props, behavior, or rendering logic.

HOC Pattern

```
function withExtraProps(WrappedComponent) {
// Returns a new component
  return function EnhancedComponent(props) {
// Add extra props or behavior
    const extraProps = { color: 'red' };

// Return wrapped component with combined props
```

```
    return <WrappedComponent {...props} {...extraProps} />;
  };
}

// Usage
const ButtonWithExtraProps = withExtraProps(Button);
```

Real-World HOC Example: withData

Here's a simpler example of a Higher-Order Component with functional components:

```
// withHover.js - A simple HOC that adds hover functionality
function withHover(WrappedComponent) {
  return function WithHover(props) {
    // Track hover state
    const [isHovered, setIsHovered] = useState(false);

    // Hover event handlers
    const handleMouseEnter = () => setIsHovered(true);
    const handleMouseLeave = () => setIsHovered(false);

    return (
      <div
        onMouseEnter={handleMouseEnter}
        onMouseLeave={handleMouseLeave}
      >
        {/* Pass the wrapped component all original props plus the hover stat
e */}
        <WrappedComponent {...props} isHovered={isHovered} />
      </div>
    );
  };
}
```

```
// A simple component that will use the hover functionality
function Button({ isHovered, children }) {
  // Apply different styles based on hover state
  const style = {
    padding: '10px 15px',
    backgroundColor: isHovered ? '#0056b3' : '#007bff',
    color: 'white',
    border: 'none',
    borderRadius: '4px',
    cursor: 'pointer',
    transition: 'background-color 0.3s'
  };

  return (
    <button style={style}>
      {children}
    </button>
  );
}

// Create an enhanced button with hover functionality
const HoverButton = withHover(Button);

// Usage in app
function App() {
  return (
    <div>
      <h1>Hover Example</h1>
      <HoverButton>Click Me</HoverButton>
    </div>
  );
}
```

This simple example:

1. Creates a HOC that tracks hover state

2. Wraps a button component to enhance it with hover detection
3. Changes the button's appearance when hovered

The advantage is that you can now add hover functionality to any component by wrapping it with `withHover`.

▼ Render props pattern

- Portals
- React.memo and optimization