# React Interview Questions for 3 Years Experience

## Core React Concepts

### Question: What is React and what are its key features?

**Answer:** React is a JavaScript library for building user interfaces, particularly single-page applications. It was developed by Facebook and is maintained by Facebook and a community of developers.

Key features include:

- **Component-Based Architecture**: UI is built from encapsulated components that manage their own state
- **Virtual DOM**: Efficient rendering through a lightweight representation of the actual DOM
- **Unidirectional Data Flow**: Data flows in one direction, making the code more predictable and easier to debug
- **JSX**: Syntax extension that allows writing HTML-like code in JavaScript
- **React Native**: Ability to use React for mobile app development

## Life Cycle Methods

### Question: Explain React component lifecycle methods and their use cases.

**Answer:** React class components go through a lifecycle of events:

1. **Mounting Phase:**
    - `constructor()`: Initialize state and bind methods
    - `static getDerivedStateFromProps()`: Update state based on props
    - `render()`: Required method that returns JSX
    - `componentDidMount()`: Run after component is mounted to the DOM
2. **Updating Phase:**
    - `static getDerivedStateFromProps()`
    - `shouldComponentUpdate()`: Performance optimization
    - `render()`
    - `getSnapshotBeforeUpdate()`: Capture information before update
    - `componentDidUpdate()`: Run after the update is committed to DOM
3. **Unmounting Phase:**
    - `componentWillUnmount()`: Clean up resources before removal

## React Hooks

### Question: What are React Hooks and why were they introduced?

**Answer:** React Hooks are functions that let you "hook into" React state and lifecycle features from function components. They were introduced in React 16.8 to:

1. Reuse stateful logic between components without complex patterns like render props or higher-order components
2. Split complex components into smaller functions based on related pieces
3. Use React features without classes, which can be confusing with `this` binding and can't be easily optimized by compilers

### Question: Explain useState and its benefits over this.setState in class components.

**Answer:** `useState` is a Hook that lets you add state to functional components. It returns a pair: the current state value and a function to update it.

Benefits over `this.setState`:

- More straightforward syntax without `this` binding issues
- State can be split into multiple state variables
- Each `setState` call is independent and doesn't require merging the state object
- Function updates (`setState(prev => prev + 1)`) are more intuitive

### Question: Explain useEffect and how it replaces lifecycle methods.

**Answer:** `useEffect` is a Hook that lets you perform side effects in function components. It serves the same purpose as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` combined.

It takes two arguments:

1. A function with your effect code
2. A dependency array to control when the effect runs

### Question: Explain useContext and how it simplifies working with React Context.

**Answer:** `useContext` is a Hook that lets you subscribe to React context without introducing nesting through Consumer components. It accepts a context object and returns the current context value from the nearest matching Provider.

### Question: Explain useReducer and when you would use it instead of useState.

**Answer:** `useReducer` is a Hook for state management based on the reducer pattern from Redux. It's an alternative to `useState` that's preferable when:

1. You have complex state logic involving multiple sub-values
2. The next state depends on the previous state
3. You need to optimize performance for components that trigger deep updates

It takes a reducer function and an initial state, returning the current state and a dispatch function.

### Question: Explain useMemo and useCallback. How do they improve performance?

**Answer:** Both Hooks are used for performance optimization:

- `useMemo` memoizes a computed value, recalculating only when dependencies change
- `useCallback` memoizes a function, preventing unnecessary re-renders in child components that receive the function as prop

### Question: Explain useRef and its common use cases.

**Answer:** `useRef` is a Hook that returns a mutable ref object whose `.current` property is initialized to the passed argument. The object persists for the full lifetime of the component.

Common use cases:

1. Accessing/manipulating DOM elements directly
2. Keeping a mutable value that doesn't cause re-renders when changed
3. Storing previous state or prop values
4. Storing instance variables (like you would use instance properties in classes)

### Question: How do React Hooks replace class components? Discuss the practical differences.

**Answer:** React Hooks let you use React features without writing class components:

| Class Feature | Hook Replacement | Notes |
|---|---|---|
| `constructor` | `useState`, function scope | Initialize state and bind methods |
| `this.state`/`this.setState` | `useState`/`useReducer` | More modular state management |
| `componentDidMount` | `useEffect(() => {}, [])` | Empty dependency array for mount only |
| `componentDidUpdate` | `useEffect(() => {}, [dep1, dep2])` | Run when dependencies change |
| `componentWillUnmount` | `useEffect(() => { return () => {} }, [])` | Cleanup function |
| `shouldComponentUpdate` | `React.memo`, `useMemo` | Optimize rendering |
| Class methods | Regular functions, `useCallback` | No `this` binding needed |
| Static `getDerivedStateFromProps` | `useState` + `useEffect` | Update state based on props |
| Context via `contextType` | `useContext` | Simpler context consumption |
| Instance variables | `useRef` | Store mutable values |

## State Management

### Question: What is prop drilling and what are ways to avoid it?

**Answer:** Prop drilling (also called "threading") is the process of passing props through intermediate components that don't need those props except to pass them down to deeper components. This can make

code hard to maintain and refactor.

Ways to avoid prop drilling:

1. React Context API
2. State management libraries (Redux, Zustand, MobX)
3. Component composition
4. Custom hooks to encapsulate shared logic
5. Higher-Order Components (HOCs)

### Question: Compare Redux and useReducer for state management.

**Answer:** Both Redux and `useReducer` implement the reducer pattern for state management, but with key differences:

| Feature | Redux | useReducer |
|---|---|---|
| Scope | Global app state | Component-level state |
| Middleware | Supports middleware (thunk, saga) | Requires custom solutions |
| DevTools | Robust debugging tools | Limited debugging |
| Complexity | More boilerplate code | Simpler, less code |
| Side Effects | Handled by middleware | Combine with useEffect |
| Performance | Optimized for large applications | Good for simpler state logic |
| Persistence | Built-in solutions available | Requires custom implementation |
| Learning Curve | Steeper | More straightforward |

## Advanced React Concepts

### Question: What are Higher-Order Components (HOCs) and when should you use them?

**Answer:** Higher-Order Components are functions that take a component and return a new component with enhanced functionality. They're used for code reuse, logic abstraction, and cross-cutting concerns like authentication or data fetching.

HOCs follow this pattern: `const EnhancedComponent = higherOrderComponent(WrappedComponent);`

Use HOCs when:

- You need to share common functionality across multiple components
- You want to separate concerns (UI rendering vs behavior)
- You need to add features like logging, access control, or performance tracking

### Question: What are render props and how do they compare to HOCs?

**Answer:** Render props is a technique where a component takes a function prop that returns a React element, allowing the component to share its state or behavior with the function.

Comparison with HOCs:

| Aspect | Render Props | HOCs |
|---|---|---|
| Implementation | Uses a function prop | Wraps component in another function |
| Prop Collision | No naming conflicts | Can overwrite props accidentally |
| Composition | More explicit | Can create "wrapper hell" |
| Debuggability | Easier to trace | Can be harder to debug |
| Reusability | Clear separation of concerns | Good for cross-cutting concerns |

### Question: What are React portals and when would you use them?

**Answer:** React portals provide a way to render children into a DOM node that exists outside the DOM hierarchy of the parent component. This is useful for components like modals, tooltips, or floating menus that need to visually "break out" of their container.

Use portals when:

1. Dealing with parent components with `overflow: hidden` or `z-index` constraints
2. Creating modals, dialogs, tooltips, or popovers
3. Working with third-party DOM libraries
4. Creating floating elements that need to appear above other content

### Question: How does React handle error boundaries? Why are they useful?

**Answer:** Error boundaries are React components that catch JavaScript errors in their child component tree, log those errors, and display a fallback UI. They prevent the entire application from crashing when an error occurs in a part of the UI.

Key points:

- Error boundaries catch errors during rendering, in lifecycle methods, and in constructors
- They do NOT catch errors in event handlers, asynchronous code, or server-side rendering
- Since React 16, uncaught errors will unmount the entire React component tree

### Question: What are React's Suspense and lazy loading features?

**Answer:** React Suspense and lazy loading are features that enable code-splitting and component loading on demand, which improves performance by reducing the initial bundle size.

- `React.lazy()`: Lets you define a component that is loaded dynamically
- `<Suspense>`: Displays a fallback content while waiting for lazy components to load

Benefits:

1. Smaller initial bundle size, leading to faster page loads
2. Better user experience with fallback UI during loading
3. Loading code only when needed (on demand)
4. Route-based code splitting for larger applications

### Question: What is the useTransition hook and how does it work?

**Answer:** `useTransition` is a React Hook introduced in React 18 that helps manage state transitions by marking certain updates as non-urgent. It allows you to keep the UI responsive during heavy state updates by prioritizing more important updates.

The hook returns a tuple with:

1. A boolean `isPending` that indicates whether a transition is in progress
2. A `startTransition` function to wrap state updates that can be deferred

Use `useTransition` when:

1. You have UI updates that are computationally expensive
2. You want to maintain responsive input fields during heavy operations
3. You need to prioritize user interactions over background state updates
4. You're implementing search-as-you-type functionality with large datasets

## Performance Optimization

### Question: What are the common techniques to optimize React application performance?

**Answer:** React applications can be optimized using various techniques:

1. **Component Optimization:**
    - Use React.memo for functional components
    - Implement shouldComponentUpdate for class components
    - Use PureComponent for simple comparison-based optimization
2. **State Management:**
    - Keep state as local as possible
    - Use appropriate state management tools (Context API, Redux, etc.)
    - Normalize complex state structures
3. **Rendering Optimization:**
    - Virtual list libraries for long lists (react-window, react-virtualized)
    - Code-splitting with React.lazy and Suspense
    - Avoid unnecessary re-renders with proper key usage
4. **Hooks Optimization:**
    - Memoize callbacks with useCallback
    - Memoize computed values with useMemo
    - Use useTransition for expensive updates
5. **Bundle Optimization:**
    - Tree shaking to eliminate dead code
    - Dynamic imports for code splitting

- Lazy loading of components and routes

6. **Other Techniques:**
   - Debouncing and throttling for frequent events
   - Image optimization and lazy loading
   - Web Workers for CPU-intensive tasks

### Question: Explain React.memo and when you should use it.

**Answer:** `React.memo` is a higher-order component that memoizes a functional component, preventing unnecessary re-renders when its props have not changed.

It performs a shallow comparison of props by default, but you can provide a custom comparison function as a second argument.

When to use React.memo:

1. For pure functional components that render the same result given the same props
2. For components that render often with the same props
3. For medium to large components where re-rendering is expensive
4. In component trees where a parent component updates frequently but child props don't change

When NOT to use React.memo:

1. For simple components where memoization overhead exceeds rendering cost
2. For components whose props change frequently
3. When most prop changes should cause re-rendering anyway

### Question: How do React keys work and why are they important for performance?

**Answer:** React keys are special attributes used to help React identify which items have changed, been added, or been removed in a list. They play a crucial role in React's reconciliation process.

Keys should be:

- Unique among siblings
- Stable across renders
- Preferably from your data (like IDs)

Performance implications of keys:

1. **Efficient Updates:** Proper keys help React identify which elements can be reused and which need to be recreated
2. **Preserved State:** Components maintain their state when moved within a list
3. **Reduced DOM Operations:** Fewer DOM manipulations when list items change
4. **Deterministic Rendering:** Predictable behavior during reconciliation

Common key mistakes:

1. Using index as key in dynamic lists (especially for lists that can be reordered or filtered)
2. Using non-stable values like random numbers generated during render

3. Using non-unique values like duplicate strings

## Testing in React

### Question: What testing frameworks and tools are commonly used with React applications?

**Answer:** Common testing frameworks and tools for React applications include:

1. **Test Runners and Frameworks:**
    - Jest (most popular, included with Create React App)
    - Mocha
    - Jasmine
2. **Testing Libraries:**
    - React Testing Library (recommended by React team)
    - Enzyme (older, less maintained now)
    - Cypress for end-to-end testing
    - Playwright for cross-browser testing
3. **Utilities:**
    - Mock Service Worker (MSW) for API mocking
    - user-event for simulating user events
    - jest-dom for custom DOM matchers
4. **Snapshot Testing:**
    - Jest's snapshot functionality
    - Storybook with Storyshots

# JavaScript Interview Questions for 3 Years Experience

## Hoisting

**Question:** Explain hoisting in JavaScript with an example.

**Answer:** Hoisting is JavaScript's default behavior of moving declarations to the top of their scope during compilation. Only declarations are hoisted, not initializations.

## Closure

**Question:** What is a closure in JavaScript and what are its practical uses?

**Answer:** A closure is a function that remembers and can access variables from its outer scope even after that scope has finished executing. Closures are commonly used for data privacy, function factories, and maintaining state.

## Scope (var, let, const)

**Question:** Explain the differences between var, let, and const with examples.

**Answer:**

- `var`: Function-scoped, can be redeclared and updated, hoisted to the top with initial value undefined
- `let`: Block-scoped, can be updated but not redeclared within same scope, hoisted but not initialized
- `const`: Block-scoped, cannot be updated or redeclared, must be initialized at declaration, hoisted but not initialized

## Undefined VS NULL VS Not Defined

**Question:** What's the difference between undefined, null, and not defined?

**Answer:**

- `undefined`: Variable has been declared but not assigned a value
- `null`: Intentional absence of value (assigned by developer)
- `not defined`: Variable that has not been declared (ReferenceError)

## Promise, async/await

**Question:** How do Promises work, and how does async/await simplify asynchronous code?

**Answer:** Promises represent the eventual completion or failure of an asynchronous operation and its resulting value. Async/await is syntactic sugar built on top of Promises that allows writing asynchronous code in a more readable, synchronous-like manner.

## Debounce

**Question:** What is debounce and how would you implement it?

**Answer:** Debounce is a technique to limit the rate at which a function can fire. It ensures that a function won't be executed until after a certain amount of time has passed since it was last called. This is useful for performance optimization, especially for expensive operations triggered by frequent events (like resize, scroll, input).

## Throttle

**Question:** What is throttling and how would you implement it?

**Answer:** Throttling is a technique that ensures a function is called at most once in a specified time period. Unlike debounce, throttle will execute the function regularly at the specified rate, rather than waiting for a period of inactivity.

## Shallow VS Deep Copy

**Question:** What's the difference between shallow and deep copy in JavaScript?

**Answer:**

- **Shallow copy**: Creates a new object/array, but nested objects/arrays are still references to the original.
- **Deep copy**: Creates a completely independent copy including all nested objects.

## Currying

**Question:** What is currying in JavaScript and what are its benefits?

**Answer:** Currying is a technique of transforming a function that takes multiple arguments into a sequence of functions that each take a single argument. Benefits include partial application of functions, better code reuse, and more readable code composition.

## SetTimeout

**Question:** How does setTimeout work and what are common pitfalls?

**Answer:** setTimeout schedules a function to run after a specified delay (in milliseconds). Common pitfalls include scope issues with `this`, non-guaranteed timing precision, and how it works in the event loop.

## Call, Apply, Bind

**Question:** Explain the purpose and differences between call, apply, and bind methods.

**Answer:**

- `call`: Calls a function with a given `this` value and arguments provided individually
- `apply`: Calls a function with a given `this` value and arguments provided as an array
- `bind`: Creates a new function with a bound `this` value and optional preset arguments

## ES6+ Features

**Question:** What are some key ES6+ features and how have they improved JavaScript?

**Answer:** ES6+ introduced many features that made JavaScript more powerful and expressive:

## Higher-Order Functions

**Question:** What are higher-order functions? Give examples.

**Answer:** Higher-order functions are functions that take other functions as arguments or return functions as results. They enable more modular, reusable, and expressive code. Examples include map, filter, reduce, and function composition.

## Pure Functions and Immutability

**Question:** What are pure functions and why is immutability important in JavaScript?

**Answer:** Pure functions are functions that:

1. Given the same input, always return the same output
2. Have no side effects (don't modify external state)
3. Don't depend on external state

Immutability means not changing data once it's created. This approach leads to more predictable code, easier debugging, better testability, and works well with React's rendering model.

## Error Handling

**Question:** How do you handle errors in JavaScript?

**Answer:** JavaScript uses try/catch blocks for synchronous code and either promises with .catch() or try/catch with async/await for asynchronous code. Custom errors can be created by extending the Error class.

## Event Loop & Asynchronous JavaScript

**Question:** Explain the JavaScript Event Loop and how it handles asynchronous operations.

**Answer:** The Event Loop is JavaScript's mechanism for handling asynchronous operations. JavaScript is single-threaded, but can perform non-blocking operations through the event loop, which constantly checks if the call stack is empty and then processes items from the callback queue.

The execution happens in this order because:

1. Synchronous code executes first ("Start", "End")
2. Microtasks (Promises) execute before macrotasks (setTimeout)
3. Even with 0ms timeout, setTimeout callback goes through the task queue

### == vs ===

**Question:** What's the difference between `==` and `===` operators?

**Answer:**

- `==` (loose equality): Compares values after type conversion
- `===` (strict equality): Compares both value and type without conversion

## Prototypal Inheritance

**Question:** Explain JavaScript's prototypal inheritance model.

**Answer:** In JavaScript, objects inherit directly from other objects through a prototype chain. Each object has an internal prototype link to another object, and if a property isn't found on the object itself, JavaScript looks up the prototype chain until it finds it or reaches the end (null).

## Function Declaration vs Expression

**Question:** What's the difference between function declarations and function expressions?

**Answer:**

- **Function Declaration**: Defined with the `function` keyword at the start of a statement, hoisted completely
- **Function Expression**: Function assigned to a variable, only variable declaration is hoisted (if using var)

## The 'this' Keyword

**Question:** How does the 'this' keyword work in JavaScript?

**Answer:** The value of 'this' depends on how a function is called, not where it's defined:

1. In global context: refers to the global object (window in browsers, global in Node.js)
2. In a method: refers to the object that owns the method
3. In a function: refers to global object (non-strict) or undefined (strict mode)
4. In an event: refers to the element that received the event
5. In arrow functions: inherits 'this' from the parent scope
6. With call/apply/bind: explicitly set by these methods

## Event Propagation & Delegation

**Question:** Explain event bubbling, capturing, and delegation in JavaScript.

**Answer:**

- **Event Bubbling**: Events "bubble" up from the target element to its ancestors
- **Event Capturing**: Events are first captured by the outermost element, then propagate to the target
- **Event Delegation**: Using event bubbling to handle events at a higher level in the DOM, reducing the number of event listeners

## LocalStorage vs SessionStorage vs Cookies

**Question:** Compare and contrast localStorage, sessionStorage, and cookies.

**Answer:**

| Feature | localStorage | sessionStorage | Cookies |
|---|---|---|---|
| Lifetime | Until explicitly deleted | Tab session | Can set expiration |
| Storage Size | ~5MB | ~5MB | ~4KB |
| Sent with Requests | No | No | Yes (increases traffic) |
| Accessibility | Any window | Same tab only | Any window |
| API | Simple key/value | Simple key/value | String parsing required |

## Web Performance Optimization

**Question:** What techniques would you use to optimize a JavaScript application's performance?

**Answer:** JavaScript performance optimization involves several strategies:

## AJAX and Fetch API

**Question:** Compare XMLHttpRequest and the Fetch API for making HTTP requests.

**Answer:** Both are used to make HTTP requests, but Fetch is modern and uses Promises, while XMLHttpRequest is older with a callback-based approach.

## Micro-optimization Techniques

**Question:** What micro-optimization techniques can improve JavaScript code performance?

**Answer:** While modern JavaScript engines are very optimized, there are still techniques that can improve performance:

## Testing Strategies

**Question:** What testing strategies and tools would you use for a JavaScript application?

**Answer:** Effective JavaScript testing involves several layers and approaches:

## Security Best Practices

**Question:** What JavaScript security best practices should developers follow?

**Answer:** Security in JavaScript applications requires attention to several areas:

## Design Patterns in JavaScript

**Question:** What are some common design patterns in JavaScript and when would you use them?

**Answer:** Design patterns are reusable solutions to common problems. Here are some important patterns in JavaScript:

**Example:**

```
// 1. Module Pattern
const calculator = (function () {
  // Private variables
  let result = 0;

  // Public interface
  return {
    add(x) {
      result += x;
      return this;
    },
    subtract(x) {
      result -= x;
      return this;
    },
    getResult() {
      return result;
    },
  };
})();

calculator.add(5).subtract(2);
console.log(calculator.getResult()); // 3

// 2. Singleton Pattern
const Database = (function () {
```

```
  let instance;

  function createInstance() {
    return {
      data: {},
      get(key) {
        return this.data[key];
      },
      set(key, value) {
        this.data[key] = value;
      },
    };
  }

  return {
    getInstance() {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    },
  };
})();

const db1 = Database.getInstance();
const db2 = Database.getInstance();
console.log(db1 === db2); // true

// 3. Observer Pattern
class EventEmitter {
  constructor() {
    this.events = {};
  }

  on(event, listener) {
    if (!this.events[event]) {
      this.events[event] = [];
    }
    this.events[event].push(listener);
    return this;
  }

  emit(event, ...args) {
    if (!this.events[event]) return false;
    this.events[event].forEach((listener) => {
      listener.apply(this, args);
    });
    return true;
  }

  off(event, listener) {
    if (!this.events[event]) return this;
    this.events[event] = this.events[event].filter((l) => l !== listener);
    return this;
```

```
  }
}

const emitter = new EventEmitter();
const handler = (data) => console.log("Data received:", data);
emitter.on("data", handler);
emitter.emit("data", { value: 42 }); // "Data received: { value: 42 }"

// 4. Factory Pattern
function createUser(type) {
  if (type === "admin") {
    return {
      name: "Admin",
      permissions: ["read", "write", "delete"],
    };
  } else if (type === "user") {
    return {
      name: "User",
      permissions: ["read"],
    };
  }
}

const admin = createUser("admin");
console.log(admin.permissions); // ['read', 'write', 'delete']

// 5. Decorator Pattern
function Car() {
  this.cost = function () {
    return 20000;
  };
}

function withAC(car) {
  const originalCost = car.cost();
  car.hasAC = true;
  car.cost = function () {
    return originalCost + 1000;
  };
  return car;
}

function withSunroof(car) {
  const originalCost = car.cost();
  car.hasSunroof = true;
  car.cost = function () {
    return originalCost + 1500;
  };
  return car;
}

const myCar = withSunroof(withAC(new Car()));
console.log(myCar.cost()); // 22500
```

## Memory Management & Leaks

**Question:** How does memory management work in JavaScript, and how can you prevent memory leaks?

**Answer:** JavaScript uses automatic garbage collection to free memory that's no longer needed. However, memory leaks can still occur in certain scenarios:

**Example:**

```
// 1. Common memory leak: Forgotten global variables
function leak() {
  leakedVariable = "I am leaked"; // Missing 'var', 'let', or 'const'
}

// Fix: Always declare variables
function noLeak() {
  const localVariable = "I am contained";
}

// 2. Memory leak: Forgotten event listeners
function setupListener() {
  const button = document.getElementById("button");
  button.addEventListener("click", function () {
    // This keeps a reference to the button even if it's removed from DOM
    console.log("Button clicked", button);
  });
}

// Fix: Remove event listeners when no longer needed
function cleanSetup() {
  const button = document.getElementById("button");
  const handler = function () {
    console.log("Button clicked");
  };

  button.addEventListener("click", handler);

  return function cleanup() {
    button.removeEventListener("click", handler);
  };
}

// 3. Memory leak: Closures holding references
function createLargeObject() {
  const largeData = new Array(1000000).fill("data");

  return function () {
    // This function keeps largeData in memory
    // even if only a tiny bit is needed
    return largeData[0];
  };
```

```
}

// Fix: Only keep what you need
function betterFunction() {
  const largeData = new Array(1000000).fill("data");
  const firstItem = largeData[0];

  return function () {
    // Only keeps reference to what's needed
    return firstItem;
  };
}

// 4. Memory leak: Circular references (less common in modern browsers)
function createCircular() {
  const obj1 = {};
  const obj2 = {};

  obj1.ref = obj2;
  obj2.ref = obj1;

  return obj1;
}

// 5. Timers that aren't cleared
function startInterval() {
  const data = { counter: 0 };

  const intervalId = setInterval(() => {
    data.counter++;
    console.log(data.counter);
  }, 1000);

  // If we don't store intervalId or clear it,
  // this keeps data in memory forever
}

// Fix: Store and clear intervals/timeouts
function betterInterval() {
  const data = { counter: 0 };
  const intervalId = setInterval(() => {
    data.counter++;
    console.log(data.counter);

    if (data.counter >= 10) {
      clearInterval(intervalId);
    }
  }, 1000);

  return function stopInterval() {
    clearInterval(intervalId);
  };
}
```

## Web Components

**Question:** What are Web Components and how would you create one?

**Answer:** Web Components are a set of web platform APIs that allow you to create custom, reusable HTML elements. They consist of Custom Elements, Shadow DOM, and HTML Templates.

## JavaScript Frameworks Comparison

**Question:** How would you compare React, Vue, and Angular? What are their strengths and weaknesses?

**Answer:** Each framework has its own philosophy and approach:

**Key Differences:**

| Feature | React | Vue | Angular |
|---|---|---|---|
| Learning Curve | Moderate | Low | Steep |
| Size | Small | Very Small | Larger |
| Data Binding | One-way | Two-way (optional) | Two-way |
| State Management | External (Redux/Context) | Vuex | Services/RxJS |
| Syntax | JSX | Templates + JSX option | Templates with directives |
| Backed By | Facebook | Community (Vue Team) | Google |
| Model | Component-based | Component-based | Complete framework |

**Example of a Counter in Each:**

**React:**

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

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

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

**Vue:**

```
<template>
  <div>
    <p>Count: {{ count }}</p>
    <button @click="increment">Increment</button>
  </div>
</template>

<script>
  export default {
    data() {
      return {
        count: 0,
      };
    },
    methods: {
      increment() {
        this.count++;
      },
    },
  };
</script>
```

**Angular:**

```
// counter.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-counter',
  template: `
    <div>
      <p>Count: {{ count }}</p>
```